THE X10ABOT MODULAR, EXTENSIBLE, ROBOTICS PLATFORM

A Thesis
Submitted in Partial Fulfilment of the Requirement for the
Degree of Master of Philosophy in Computer Science

of
The University of the West Indies

by
Rohan Anthony Smith
2017

Department of Computing
Faculty of Science and Technology
Mona Campus

ABSTRACT

The X10ABOT Modular, Extensible, Robotics Platform

Rohan Anthony Smith

We present the architecture and design of a general purpose robotics development kit, called X10ABOT, which aims to facilitate the rapid development of robotic solutions to a wide variety of problems. Robotics kits of this sort are usually aimed at casual hobbyists and children. As a result, these kits usually have severe limitations in the number of sensors and actuators that they can manage. In contrast, X10ABOT is: modular (new functionality can be added through standard daughterboard modules), scalable (up to a maximum of 16 sensors and actuators on each of up to 112 daughterboard modules if implemented on harware with sufficient resources), and extensible (modular software can be added to support new types of sensors and actuators). We have also designed generic ports, capable of handling several types of sensors and actuators. We demonstrate the application of the platform to the development of a general purpose robotics kit, using the Arduino development board. In this implementation, we will show how the system would allow its users to readily add extra sensors and actuators to a project, with minimal configuration, and relatively little impact on the pre-existing user code.

Keywords: Robotics; Modular; Extensible.

ACKNOWLEDGEMENTS

I would like to acknowledge the support of my close family and friends who served as the motivating factor that propelled me to finish.

I would like to offer my sincere thanks to my supervisor Prof. Daniel Coore, who has provided me with endless guidance and support. He has been an inspiration in my field and has provided me with insight and vast knowledge to help in the successful completion of this thesis.

Thanks to the Department of Computing, for providing me with the opportunity to further my studies and financial support through the Departmental Award.

To the members of my research group, I owe many thanks for the drive and motivation in completing my studies. The weekly meetings are an essential part of providing the momentum to complete.

Finally, this project could not have been complete without the strength and knowledge endowed upon me by God.

DEDICATION

This thesis is dedicated to my beloved wife Valene, son Rojé and family. They have been there for me, even when I have all but given up. Thanks for keeping me going.

TABLE OF CONTENTS

LIST OF FIGURES

List of Tables

CHAPTER 1
INTRODUCTION

1.1 Context

The X10ABOT robotics platform is a general purpose robotics kit that is designed to be a low cost, modular, scalable and extensible option for developing custom robotics projects. The system's intended audience and applications range from simple primary school level robots to more complex creations used at university level robotics competitions and research programs. The X10ABOT kit's focus is to improve upon current general robotics kit standards by providing:

- Support for potentially hundreds of sensors and actuators.
- A modular design with pluggable add-on peripheral boards.
- An extensible software and hardware architecture that easily supports new devices.
- A low overall cost while achieving all the above listed requirements.

Although many general-purpose robotics kits exist, many are either quite limited in the range of models that can be built with them, or demand a high price for their expressiveness. Many of the current robotics kits are designed with a limited number of available ports for attaching sensors and actuators. This is evident in the general purpose LEGO Mindstorms Robotics kit and

most of the wheeled platforms like the Boe-bot and the Rug Warrior Pro. The underlying designs of these kits only partially support scaling, or not at all: one is forced to limit the features of one's robotic designs to operate within this limitation. Inadequate peripheral interfaces can curb creativity and innovativeness, resulting in unfulfilled designs, especially for robotics novices.

The `X10ABOT` robotics platform is a mix of both hardware and software. The hardware consists of two types of modules, a motherboard and daughterboard. These however can be implemented on a single piece of hardware with a single microprocessor or on physically separate hardware modules. For physically separate boards, they are connected via a bus that passes data and instructions between a single motherboard and one or more daughterboards. The daughterboards in turn process these instruction and data and subsequently execute them by applying them to sensors and actuators connected to their respective ports. Daughterboards can also interrupt the motherboard if they have information relevant to an active process.

There are specialised software components present on each hardware module. The motherboard hosts the user generated program code and the system software that processes it. The middleware (see Section 2.1.3.1) is present on both the motherboard and daughterboard, and is responsible for seamless cross-board communication. Finally there is the firmware which is embedded in the daughterboard, this is semi-permanent software that is specialised to interpret and execute instructions passing between the sensors and actuators to and from the motherboard.

1.2 Motivation

The motivating factors for this project came from the need for involving robotics in education. That lead to the creation of the X10ABOT robotics platform. The X10ABOT project is aimed at creating a platform that will adequately provide a full featured robotics framework for students, while maintaining a price point that is lower than the current market price for a full featured robotics kit. This will further assist in the commoditization of robotics kits and subsequently increase the possibility for activities such as robotics competitions between schools. Other driving motivations for the project include the creation of a platform that will provide a flexible starting point for other robotics research projects. The kit will also be useful enough for hobbyist users who are interested in robotics and automation, the X10ABOT robotics platform will be an attractive tool that can be utilized in their projects.

We researched the structure of several robotics research projects and commercial robotics kits and identified a number of design objectives that were useful for a general purpose architecture. We aimed at creating a robotics framework that was able to support more sensors and actuators than most of the popular robotics kits, this framework had to support most of the common off-the-shelf sensors and actuators and yet be easy to use without limiting creativity.

We reviewed many types of robotics projects and came across a few interesting concepts. They influenced and validated the design decisions made in the X10ABOT project. We have reviewed a few of them and present an overview of their architectures in Chapter 2.

1.3 Thesis Summary

Outlined in this thesis are the details of how we progressed from our conceptual ideal product of a low-cost, modular, scalable, extensible robotics kit to transforming these ideals into realistic software and hardware components. These components were modelled and tested using the Arduino embedded development platform. We utilized one or more Arduinos to represent the motherboard and daughterboard setup that we envisioned. We also took advantage of the Arduino software framework which was used to develop the system software, middleware and firmware.

The outcome of the use of these tools and strategies gave us a system that was able to easily control multiple sensors and actuators in a scalable and modular way while allowing for extensibility.

Figure 1.1 describes a typical code setup for a very simple mobile robot using the `X10ABOT` framework. All the sensors and actuators are placed on one board (the daughterboard) for this simple implementation. This robot has one actuator (a DC motor connected to an external H-bridge), a force sensor and a push-button sensor. The instructions written in Figure 1.1 drives the motor at full speed whenever either the touch sensor records an input or if there is a force on the force sensor above a certain threshold, otherwise the motor will remain in the off state.

We then further extend the same simple example from Figure 1.1 to show how the `X10ABOT` framework is able to scale modularly. In Figure 1.2 an extra DC motor and a light sensor is added as a single module. Instructions are then added to activate the new DC motor if there is a light intensity

Simple Example

```
/**
* Import the necessary libraries for the motherboard,
* internal daughterboard and the peripheral bus
**/
#include <Wire.h>
#include <X10ABOT_MB.h>
#include <X10ABOT_DB.h> //Include the internal daughterboard #0 (SELF)

//Initialise the DC motor on daughterboard #0 (SELF), actuator port #1
Actuator motor1(SELF,1);
//Initialise force sensor on daughterboard #0 (SELF), sensor port #1
Sensor force1(SELF,1);
//Initialise force sensor on daughterboard #0 (SELF), sensor port #2
Sensor pushbutton(SELF,1);

void setup(){}
void loop(){
//Continuously check the sensors for a reading
  if(pushbutton.readDigitalB() || (force1.readAnalog()>100)){
    motor1.setDigital(LO, HI); //Turn motor1 on by sending LO on pin a and HI on pin B
    motor1.pwm_a(100); //operate motor1 at full power (100%)
  }else{
    motor1.setDigital(LO, LO);//Turn motor1 off by sending LO on pin a and pin b
  }
}
```

Figure 1.1. Example of a single controller with a force sensor and a DC motor

above a certain threshold on the sensor. The code example shows that even with the addition of two independent devices, *motor1* and *lightSensor* to the system, it did not affect the existing code but fit seamlessly in the development process.

Thesis Statement *It is possible to design an architecture for a general purpose robotics platform using the Arduino™ microcontroller and supporting software, open-source libraries and tools. It appears that by using a modular, distributed architecture for both hardware and software, it is possible to create a low cost, scalable system with the ability to support hundreds of sensors and actuators. Furthermore, the same API for hardware control that is used to implement the modular and scalable design can be leveraged to allow user defined extensions to hardware interfaces. Finally, all of this can be implemented with only moderate latency penalties (250-540µs per digital read/write) .*

Complex addition to simple example

```
/**
* Import the necessary libraries for the motherboard,
* internal daughterboard and the peripheral bus
**/
#include <Wire.h>
#include <X10ABOT_MB.h>
#include <X10ABOT_DB.h> //Include the internal daughterboard #0 (SELF)

//Initialise the DC motor on daughterboard #0 (SELF), actuator port #1
Actuator motor1(SELF,1);
//Initialise force sensor on daughterboard #0 (SELF), sensor port #1
Sensor force1(SELF,1);
//Initialise force sensor on daughterboard #0 (SELF), sensor port #2
Sensor pushbutton(SELF,1);

void setup(){}
void loop(){
//Continuously check the sensors for a reading
if(pushbutton.readDigitalB() || (force1.readAnalog()>100)){
    motor1.setDigital(LO, HI); //Turn motor1 on by sending LO on pin a and HI on pin B
    motor1.pwm_a(100); //operate motor1 at full power (100%)
  }else{
    motor1.setDigital(LO, LO);//Turn motor1 off by sending LO on pin a and pin b
  }
 const byte BOARD2 = 9; //Constant with arbitrary daughterboard address
 //Declare motor on daughterboard #9, actuator port #1
 Actuator motor2(BOARD2,1);
 //Declare force sensor on daughterboard #9, sensor port #1
 Sensor lightSensor(BOARD2,1);

//Added extra functionality with a new sensor and a new actuator
//on daughterboard #9 while light is on the sensor, activate motor2
if (lightSensor.readAnalog()>700)
  {
    motor2.setDigital(HI, LO); //Turn motor2 on by sending HI on pin A and LO on pin b
    motor2.pwm_a(50); //operate motor2 at half power (50%)
  }else{
    motor2.setDigital(LO, LO);//Turn motor1 off by sending LO on pin a and pin b
  }
}
```

Figure 1.2. Now we add: A push-button sensor to the motherboard (or daughterboard #0). We also add an extra daughterboard with a light sensor, and a DC Motor

1.4 Structure of Thesis

Chapter 2 explores the design goals of the `X10ABOT` platform, how the same principles are being used by other robotics systems. Chapter 3 describes the design and implementation of the entire platform, the technologies used and how we integrated them to work seamlessly together. Chapter 4 illustrates the results that demonstrates the effectiveness of the design strategies we chose. Chapter 5 explores the potential areas for future work and concludes the thesis.

CHAPTER 2
BACKGROUND AND MOTIVATION

Robotics development is a continually growing field with a promising future. The current state of robotics can be compared to that of the PC industry in its embryonic years (Gates 2007). Many of the issues that the PC industry faced are redefining themselves in today's robotics arena. The most popular ones include high costs, a lack of common standards and very few generic platforms. Whenever it becomes necessary to develop a new robot for a particular task, it is often built from scratch. This is because there are few general purpose platforms that meet all the needs of the typical robotics engineer, student and hobbyist (Bonarini et al. 2012). The "killer app" for the robotics industry will be a platform addresses these issues and in doing so, create a broader adoption of robotics technology.

As with the computer industry, one of the main markets that utilize robotics is education. Robotics is used to teach students of all ages and levels on topics ranging from science to social skills (Kramer and Scheutz 2006). General Purpose Robotic kits are one of the many 'edutainment' methods that are developed. These kits provide the user with the ability to build robots for a large range of applications. An increased interest in robotics and robotics development has fuelled the search for a robotics platform that will be useful for schools and the typical hobbyist. There has been research into finding a suitable platform that is flexible, inexpensive, extensible and capable of handling projects of varying complexities (Weiss and Overcast 2008). Popular kits exist that meet some of these characteristics. For

instance, the Lego Mindstorms™robotics kit is one of the most popular robotics platforms that can achieve many of these requirements. It however achieves a few of them through unconventional 'hacking' of the platform, or by purchasing third party equipment. This is not always easy to accomplish for novice developers.

"Studies show that robotics generates a high degree of student interest and engagement and promotes interest in math and science careers . The robotics platform also promotes learning of scientific and mathematic principles, through experimentation" (Barker and Ansorge 2007). The inclusion of robotics in a school curriculum has been shown to improve performance in STEM subjects (Benitti 2012). Robotics development enables students to apply practical techniques to complete tasks and to solve scientific problems. The result or response from their work is immediately observable through the physical response of the robot. In addition, robotics presents a 'fun factor' that attracts a lot of young people.

2.1 Similar Projects

There are numerous robotics platforms, both custom built and general purpose that implement useful design strategies. The following outlines many of the core concepts found in these projects and the advantages of their approach.

The Tetrixx robotics project was developed in the late 1990's. It sought to overcome the issue of expandability being faced by its contemporaries such as MIT's 6.270 board and Handyboard (Martin 2000) projects. To overcome this limitation, the Tetrixx project employed specialised sensor and

actuator expansion boards connected over a bus. The bus supported up to 64 expansion boards and allowed the Tetrixx kit to manage far more peripherals than supported by the connectors from one single controller (Enderle et al. 2000).

Another interesting robotics kit project is the LEGO Mindstorms NXT robotics development platform. This is a very popular robotics kit among schools and hobbyists. It has three actuator ports and four sensor ports on its controller unit. Additional sensors and actuators can be added either by pairing with another controller or by purchasing third party multiplexing devices that allow multiple peripheral devices connected to one port. The port design is quite extensible with numerous analog and digital I/O pins and an I^2C bus to support even very complex devices. The Mindstorms kit has the distinct advantage of having a very active community of developers who have worked to create a large set of programming interfaces that can be used to write robotics applications for the platform. These include compilers for some of the most popular programming languages such as, C, Java, Python, Ruby, Ada, Lua and Matlab. The LEGO Mindstorms kit uses a very modern architecture and is as a standard by which many robotics kits are measured. The LEGO Mindstorms kit however lacks ease of scalability some users require. It is among one of the only commercial robotic kits that is open sourced, however its cost is prohibitive for many thus hindering large-scale adoption.

The architecture of the Hannibal Hexapod robotics project has a commendable capability of managing the input and output signals for over one hundred (100) physical sensors and actuators (Ferrell 1995). The Hexapod robot controller is based on the Subsumption Architecture and is fully

distributed across eight(8) on-board computers: one each for the main controller, the robots body and each of its six legs. All these modules are connected via an Inter-Integrated Circuit (I^2C) serial communication bus. These components were the realisation of the Hannibal Hexapod's design requirements of being scalable, modular, flexible, robust and adaptive.

The Rapid Robot Prototyping(R2P) project design goals were to be an inexpensive, open source, modular architecture for rapid prototyping of robotic applications (Bonarini et al. 2012). This architecture is aimed at students and hobbyists creating robotic applications on low powered microcontrollers. The architecture is built around a distributed system of modules communicating over a high speed serial data CAN bus. It emphasises its plug and play capability by providing ports on each module to support daisy-chaining of multiple peripheral boards. On the software aspect, the R2P has embedded firmware on its peripheral modules to act as a Hardware Abstraction Layer(HAL) to encourage modularity. On the main controller, it implements a Real Time Operating System(RTOS), a publish/subscribe middleware and a Virtual Machine that supports an embedded scripting language. The CAN serial bus is used for inter-module communication.

2.1.1 Modularity

Modularity in the design of a robotics kit architecture defines the capability to incrementally add fully independent sub-systems to a robotics project without significantly modifying the existing configuration. According to Oraw and Tinder (Oraw and Tinder 2004), "The modularity of a robot

is demonstrated by its expandable intelligence. Sensory modules that implement the robot's data protocol can be plugged into the system without reprogramming the original components". They also mention that the key to modularity is the use of peripheral modules.

Many robotics projects opt to use a distributed system architecture. This enables the control of the overall system to be shared across multiple independently operating modules. These modules operate efficiently because there is very minimal usage of shared resources, this is because each module is usually self sufficient and simply passes information back and forth between itself and the main controller over the data bus. In a centralised system which is dependent on one main processor, management of peripherals can become complex when handling a large number of sensors and actuators. The controller's resources and connections are finite and there may be contention for these resources. This can prevent the inclusion of additional modules without needing to remove a previously installed module or somehow hacking the system. Using a distributed system not only reduces complexity but it also allows for changes to be made across the entire architecture without affecting the existing setup (Avcı 2008).

A subsequent advantage of a modular architecture is software and hardware reuse, and rapid prototyping. Peripherals can be incrementally added to build more complex projects and subsequently reduce the overall cost of development (Bonarini et al. 2011). Different functions of the system can be delegated to independent and specialised modules. This will simplify the control problem by decomposing the global task into local tasks for the robot's subsystems (Gerkey and Conley 2011). Making use of modularity

reduces the computational load on the main processor by delegating the majority of the low level sensor and actuator processing to sub-modules.

A common practice that many robotics architectures employ is the inclusion of peripheral sensor and actuator boards in their design. This can be seen in the Tetrixx project (Enderle et al. 2000) and numerous other robots built around a distributed processing architecture. Peripheral boards are secondary computational units that are responsible for low level control operations. These peripheral boards interpret and execute data and control commands communicated between the main controller and the board's sensors and actuators. Peripheral boards are connected to a peripheral bus which will be able to uniquely identify each module and by extension each sensor and actuator along the communication line.

2.1.2 Scalability

Robotics kits that can support projects of varying complexities should have the ability to support as many sensors and actuators as may be required for a complex robotics project. A robotics project can range from robots that complete simple tasks that require one or two sensors and actuators to robots that manage tens or even hundreds of sensors and actuators necessary to complete their task. In general, as the complexity of a robotics project increases, so does the number of peripherals required to complete the tasks. Projects that may solve reasonably technical tasks will sometimes have to improvise by finding creative ways to use available resources. However, there may be times that the scale of the project requires a large number of sensors and actuators to adequately perform the task. Robots that try to replicate animal or human ability are usually known for using

a large number of sensors and actuators. One such example is the Hannibal Hexapod Robot. Hannibal receives a tremendous amount of sensor information while continually controlling its almost 20 servo motors. The spider-like robot has over 60 sensors of different types (Ferrell 1995). Other types of robots that utilize many sensors and actuators include robotics arms, full humanoid robots, and animal replicas like MARS: the Modular Autonomous Robotics Snake (Oraw and Tinder 2004). In all of the above examples, the supporting architecture facilitated the inclusion and efficient management of a significant number of sensors and actuators.

A distributed robotics architecture is a major contributor in decreasing the cost-to-scale factor. In a distributed system, the processing is shared between a central processing unit or a motherboard and numerous specialised modules. These modules are usually focused on doing simple tasks and therefore will not require complex hardware or devices to accomplish them. This can therefore allow them to to be inexpensive and simply connect to the motherboard to report the results of their computations and fetch data on request.

Proper power management is very important when managing tens to hundreds of sensors and actuators. These devices can behave erroneously or simply fail to perform if they are not supplied with adequate power. It is also very important to place electrically noisy and high powered devices on separate power sources. This is especially true with motors and sensitive, low powered electronics, these typically should not share the same power source. This can electrically damage sensitive components of a controller and render the robot unusable. It is also necessary that the system be sized for its maximum power requirements and a capable power source selected.

Under-powering a robot can also cause unpredictable results. Sometimes it may even be necessary to have multiple power sources on a robot that are adequately isolated.

Another requirement that may be easily overlooked is a structured and organised wiring and connection system. A robotics project can become increasingly difficult to develop if there is a web of wires and connections attached to the dozens of devices that may be present on a robot. A design that will mitigate this issue is the use of a distributed system supported by a main bus with an addressable serial data connection. If the architecture is designed with a single peripheral bus as its communication backbone that connects all devices, most of the wiring can be placed in a single organised package. The advantage of using an addressable serial bus is that these usually utilize very few connection lines. They also allow for new modules to be daisy chained onto each other. Any addition of a new set of sensors or actuators will simply equate to physically plugging a single set of wires from one module to another.

A very popular communication interface encountered in multiple projects was the Inter-Integrated Circuit (I^2C) bus. I^2C is a multi-master, addressable slave, two wire bus serial protocol. It supports up to 112 uniquely addressable devices per bus (128 minus 16 reserved addresses), 8-bit oriented, bidirectional data transfers can be made at 100 Kbits/s in the Standard-mode, 400 Kbits/s in the Fast-mode, 1 Mbits/s in Fast-mode Plus, or up to 3.4 Mbits/s in the High-speed mode (Mirats Tur and Pfeiffer 2006; Himpe 2015). While operating in its standard-mode, the I^2C bus can be as long as 9 - 12 feet without any significant noise interference (Ferrell 1995). This distance is quite sufficient for a robotics kit where the peripherals are

not usually more than 3 feet away from the controller circuit. As such, I^2C provides a low-cost solution for interconnecting large numbers of devices operating at relatively slow speeds, such as sensors or other external devices connected to a microcontroller. I^2C inherently supports collision detection and bus arbitration, bus arbitration refers to a portion of the protocol that ensures that when multiple masters try to control the bus simultaneously, I^2C allows only one master full control of the bus while queuing the prospective master without any corruption or data loss (Semiconductors 2007). A number of robotics projects implement this method of data communication, including the popular NXT LEGO Mindstorms Kit, ISocRob, and the Hannibal Hexapod autonomous robot (Ferrell 1995; Ventura et al. 2000).

2.1.3 Extensibility

A truly extensible robotics system permits future addition and support of new sensors and actuators that might not have been included in the initial design. Extensibility of a robotics kit provides developers with the freedom to define the hardware configuration with respect to the available types of sensors and actuators (Gerkey and Conley 2011). A robotics kit may never be able to guarantee support for all types of sensors and actuators, but a truly extensible architecture will provide the means for developers to configure support for their particular hardware.

2.1.3.1 Middleware

Bakken et al. (Bakken 2001) defined middleware as follows: "a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems. It is defined as a layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system." Middleware provides abstraction and seamless communication between modules in distributed robotics system. The middleware handles communication between devices without exposing the details of the protocol or its requirements and the details of the systems hardware on either end. The middleware is used to determine what type of communication medium and protocol is appropriate and creates a standard platform independent interface for interaction. In other words, the middleware just simply presents a standard interface that communicates data and instructions. In a review of popular robotics middleware, Kramer (Kramer and Scheutz 2006) cites some major advantages of using middleware. He states that it aids in software modularity. The middleware allows for pluggable libraries that are ignorant of the actual hardware implementations. This accomplishes the platform independence and portability, Kramer found these to be common attributes of most robotics middleware.

2.1.4 Economics

An important requirement that will permit wider adoption of robotics kits is ensuring that the kit's cost and its cost to scale are kept relatively low. The cost of electronics is a decreasing function of time, however these prices

are not always instantly obvious in the field of robotics. It is almost an accepted fact that robotics is an expensive field, the prices of robotics kits range from hundreds to thousands of U.S. dollars. This is a major point of restriction, robots are expensive and often beyond the resource of many persons and even universities and organisations that want to delve into the field of robotics (Yang et al. 2008). In order to make a robotics kit that will be an attractive option for hobbyist roboticists and school programmes even as low as the elementary school level, it has to be available at a competitive price point. A very popular platform for doing robotics is the Arduino development board which at the time of this publication ranged from U.S. $15 USD to $70 USD. The prices are based on type of Arduino, place of purchase and whether the Arduino is a genuine model or a clone. The Arduino development board is not specialised for robotics but has all the major tools necessary to build a functional robot if the developer is skilled in software development and electronics. The Arduino is also one of the most popular development platforms at robotics competitions. The Arduino board also has a distinct advantage of being open-source. This gives the developer the advantage of having access to free software upgrades and longer hardware support. Being open-source also means that a skilled developer will be able to make modification to the system's code and even build their own hardware replica. These advantages can mean significant savings if a solution can be accomplished by the developer instead of purchasing an upgraded version or proprietary hardware. Arduinos have proved to be one of the best options with respect to price but requires some amount of technical expertise to build operational robots. On the other hand, another system, comparable in popularity to the Arduino, that is currently being used many in schools and by hobbyist is the Lego Mindstorms robotics kit.

The controller for the LEGO Mindstorms kit without sensors and actuators starts at $250. However it is much easier to get started building interesting robotics projects.

CHAPTER 3
THE X10ABOT ARCHITECTURE, DESIGN AND IMPLEMENTATION

In this chapter we describe how we designed the components of the X10ABOT robotics framework. We applied many of the design strategies that were introduced in chapter 2 in an effort to develop a state of the art robotics architecture. We extracted the most compelling features and combined them to create a modular, scalable and extensible system with the following components:

1. The Motherboard
2. The Daughterboard
3. The Peripheral Bus
4. Input and Output Ports
5. The Motherboard Library
6. Daughterboard Firmware
7. The Middleware

Our research influenced and validated many of our design decisions for the components of the X10ABOT architecture. We chose a combination of the best features of these technologies which we found to improve modularity, scalability and extensibility and combined them to construct our

framework. It was also important that we kept the cost of implementation relatively inexpensive. These attributes represented common design patterns observed over a number of robotics projects, either for custom robots or as a part of the architecture for general purpose robotics kits. The actual implementation of these concepts included an I^2C powered peripheral bus, separate modules to manage additional sensors and actuators, a middleware architecture, full featured sensor and actuator ports, just to name a few. We needed a robust architecture that provided more than just an increase in connection points for sensors and actuators. It was critically important that there was adequate hardware and software support for the large number of sensors and actuators.

The `X10ABOT` framework is an architecture that is somewhat platform independent. However for the purpose of demonstration, we chose to implement it on the Arduino development board. Most of the concepts can be implemented on any modern microcontroller with a high speed serial interface and adequate I/O ports comparable to what is offered by the chips used in a standard Arduino. With a proper shield built on an Arduino running the `X10ABOT` software, the entire controller would cost as little as $30 USD. This would make the `X10ABOT` a comparable kit when compared to the LEGO Mindstorms with respect to the number of ports for sensors and actuators. Boasting the additional capabilities of cost effective scalability, modularity and extensibility. These features are not easily attained with a simple Arduino kit, the framework transforms the lowly Arduino into a capable robotics platform.

3.1 The Architecture

Based on the design requirements listed above, we created an architecture that captured all the previously listed features without making any significant compromise on any. The `X10ABOT` architecture implements previously outlined requirements at both the hardware and software level. It is comprised of a motherboard which is the main central controller. The motherboard is also the head of a distributed system of peripheral sensor and actuator boards called daughterboards. The daughterboards carry out a standard set of computations synonymous to a thin client, all connected across the central peripheral bus. The motherboard coordinates and controls all operations on the entire system and all operations are either initiated or terminated at the motherboard, except in the special instance when a daughterboard temporarily takes on the role of a motherboard for interrupt management. The motherboard hosts the main program for the robot application, the middleware and the extensible libraries to support various types of sensors and actuators. Each daughterboard is equipped with firmware to interpret and execute instructions given to it by the motherboard. Each daughterboard also provides multi-faceted sensor and actuator ports that can support a broad range of sensors and actuators.

3.1.1 The Hardware Architecture

The hardware aspect of the `X10ABOT` framework consists of three major components. These are the motherboard, daughterboard and the peripheral bus which connects them all. Below we will describe in detail how each

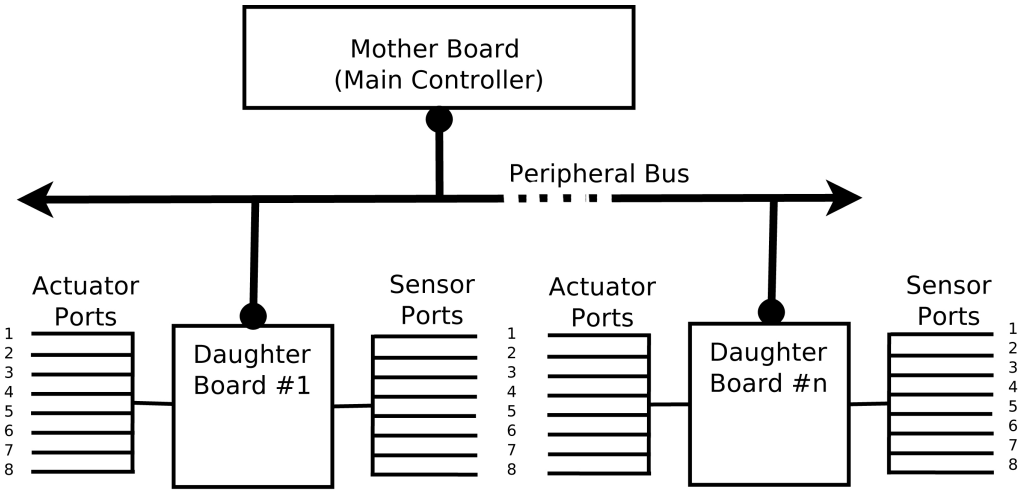

Figure 3.1. The components of the `X10ABOT` Architecture

component functions and how they relate to each other in accomplishing the goals of the architecture.

3.1.1.1 The Motherboard

The motherboard is a physical component of the robotics platform and operates as the single central processing unit. It controls the flow of data and instructions across the entire `X10ABOT` architecture. All robot control instructions written by the robotics developer are hosted on the motherboard. The instruction logic is processed on the motherboard, however the actual sensor or actuator operations are dispatched to daughterboards

as microcode instructions. These microcode instructions are then subsequently interpreted and their operations executed by sensors and actuators. As the main computational entity in the `X10ABOT` architecture, the motherboard is responsible for executing most of the decision logic, heavy computing tasks, actuating the output devices and interpreting information from input devices based on its preprogrammed set of instructions. The motherboard acts as the head of the distributed system of peripheral boards. Sensors and actuators are hosted by daughterboards which are connected to the motherboard via the peripheral bus.

3.1.1.2 The Peripheral Bus

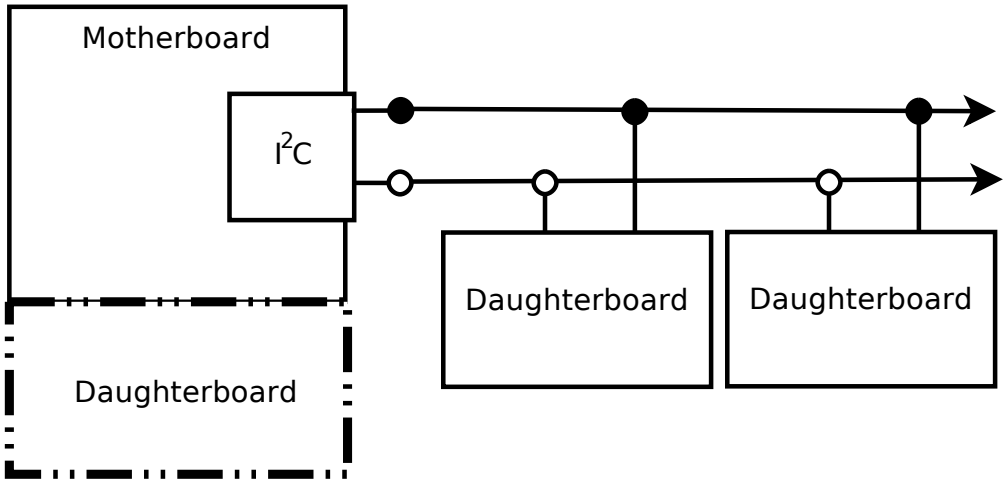

Figure 3.2. The serial communication peripheral I^2C bus

The Peripheral Bus is the central medium that connects the motherboard to one or more daughterboards and facilitates the communication of data and instructions between them. The peripheral bus is implemented using the I^2C serial communication protocol.

All communication on the I^2C bus is initiated by the bus master or, as in our case, the motherboard. This is not suitable in all situations, e.g.: when a sensor detects an input that needs to be processed immediately. Even if all the daughterboards are periodically polled, the elapsed time may cause the sensor data to become invalid or ineffective. That is why we utilized the multi-master capability of the I^2C peripheral bus to accomplish arbitration. The motherboard has primary control over the peripheral bus but can give up the control to any daughterboard that wants to gain temporary bus-master status.

Bus arbitration by the daughterboard is trivial and is accomplished by simply importing both the functionality for the motherboard and daughterboard. Each daughterboard would then have to know the motherboard's I^2C address and make the same function calls a motherboard would for the purpose of sending data when it is ready to transmit.

The peripheral bus allows for easy connection of additional devices because they can be daisy-chained onto each other reducing the need for extensive and confusing wiring. Since the peripheral bus is an I^2C bus, it is compatible with "non-daughterboard" I^2C compatible devices. The framework was designed to accommodate sensors and actuators as specialised, single device daughterboards. This was also a method of ensuring maximal extensibility.

There is one exception to the use of the I^2C bus as the main peripheral bus, this involves the use of an internal daughterboard. We will expound on this concept in the next section.

The peripheral bus is not used to transmit power from the motherboard.

Each daughter board is meant to be individually powered due to the possible varied power requirements of its resident sensors and actuators. This also makes scalability far less complicated. There needs to be no consideration of the wide range of possible power requirements.

3.1.1.3 The Daughterboard

The daughterboard is the secondary computing device in the `X10ABOT` robotics architecture. It hosts all the sensors and actuators that carry out the tasks of each robotics project. Although the daughterboard is a second tier processing component in the X10ABOT architecture, its operation is not trivial. The daughterboard is given the major task of interpreting instructions sent from the motherboard over the peripheral bus and executes them with sensors and actuators. Sensors and actuators are connected to sensor and actuator ports respectively and respond to the assertions placed upon them by the daughterboard. Daughterboards carry out simple low level operations and won't require a significant amount of processing power. There are however a few basic features that every daughterboard must possess. All daughterboards must be able to communicate over an I^2C bus and do basic digital and analog input and output. These features are very common in most microcontrollers and can be implemented without a significant cost or effort. As a member of the peripheral bus, each daughterboard must possess a unique address that the developer is aware of, it also must have knowledge of the motherboard's address. A daughterboard will need the motherboard's address whenever it has time critical data that needs to be sent to the motherboard. Using bus arbitration, the daughterboard tries to temporarily claim the bus-master role in order to quickly pass its data

to the motherboard, it would then immediately switch back to its original role.

3.1.1.4 Internal Daughterboard

On a typical motherboard circuitry, the only external I/O pins that are generally utilized are SCK and SCL that operate the I^2C protocol. This leaves a large number of unused IO pins that could have otherwise been used for some kind of operation. We then came up with a method for utilising these extra resources without breaking the design of the architecture. We designed a virtual, internal daughterboard, that uses the same microprocessor and physical board as the motherboard. This internal daughterboard is assigned with address number zero(0). All its data and instructions are communicated using direct parameter passing instead of across the peripheral bus. This particular exception is abstracted by the middleware described in section 2.1.3.1 and is totally hidden from both the motherboard and the internal daughterboard.

We do not integrate power as a part of the peripheral bus, each daughterboard is independently powered and the ports are designed to allow for each connected device on the daughterboard to use their own isolated power source.

3.1.1.5 Sensor & Actuator Ports

All sensors and actuators connect to daughterboards to send and receive data through their respective sensor and actuator ports. Ports are specially designed interfaces that were made to facilitate various types of sensors and

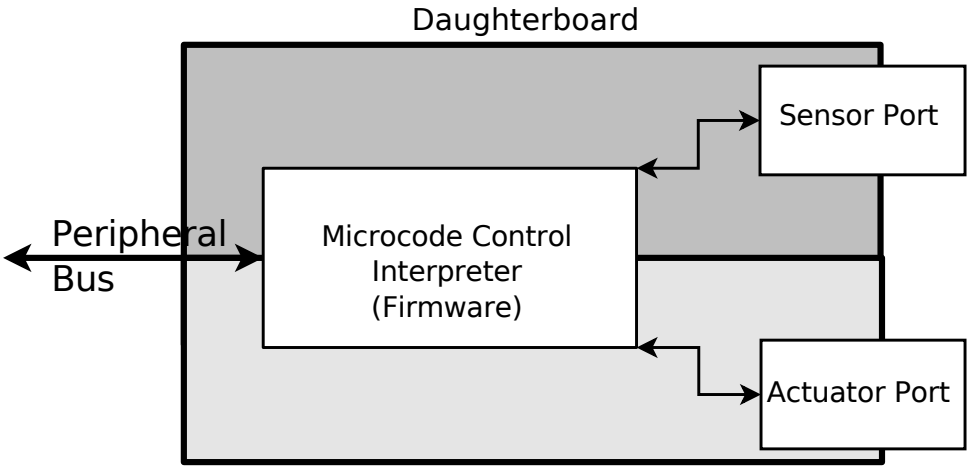

Figure 3.3. The daughterboard slave module

actuators. They were built by considering the most common technological requirements of popular sensors and actuators. The sensor and actuator ports are made up of five(5) and six(6) pins respectively.

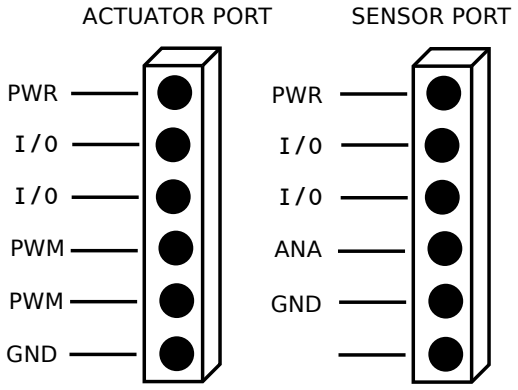

Figure 3.4. The sensor and the actuator ports

A sensor port is comprised of five(5) pins: power, ground, an analog input, and two digital I/O pins. The analog input pin is able to read voltages from analog sensors e.g. touch, light and sound sensors. The digital I/O pins can support active and passive digital sensors such as ultrasonic range-finders, passive infrared(PIR) sensors and Radio-frequency identification(RFID) sensors. The power and ground pins provide electrically

isolated power for the sensors (typically 3 or 5 volts, but in principle the power could be different from the microcontroller's voltage requirements).

An actuator port is comprised of 6 pins: power, ground, two (2) Pulse Width Modulation (PWM) output pins and two (2) digital I/O pins. The PWM signals allow support for servo motors as well as for adjusting power levels on output devices. The digital I/O pins are able to drive DC motor H-bridges or to be read as digital encoder signals. Like the sensor ports, the power supply for output ports is also electrically isolated from that of the microcontroller. This is particularly necessary when powering motors to prevent damage to the controller and other electronics.

A power bus runs through each daughterboard, connecting all the power pins of its ports. This power bus is meant to supply an alternate power source to the attached sensors and actuators. This alternate power source may or may not be suitable for powering the internal circuitry on the daughterboard, therefore, it is properly isolated. A hardware implementation of the power bus can include a jumper device to switch the power source from the basic 5V supply provided by the system to an external supply at the power specification of the sensor or actuator.

3.1.2 The Software Architecture

The `X10ABOT` hardware is managed by software distributed across both the motherboard and daughterboard. The software is responsible for the efficient operation of all components involved in the system. The software managing the architecture can be classified into four categories: firmware,

middleware, device libraries and application software. The application software, libraries and middleware reside on the motherboard while the daughterboards host the firmware. Unlike the hardware architecture, we will be describing the software with direct relation to our particular implementation on the Arduino development board.

3.1.2.1 The Motherboard Library

The motherboard robotics library is a tool-chain of functions written specifically for robotics applications. We defined and developed an extensible library of useful robotics operations for common devices and general sensors and actuators. They can be composed to control the behaviour of a robot. Our particular implementation made use of the Arduino platform which supports the addition of third-party libraries. This allows for robotics developers to take advantage of the usability of the Arduino development platform while capitalising on the usefulness of a robotics-specific set of operations implemented in the library. The library was designed not only to provide common robotics functionality to its users but to also make programming large robots with many sensors and actuators less of a hassle especially when taking advantage of the scalability and modularity of the entire platform.

3.1.2.2 Application Software

Code written by users that takes advantage of the features of the `X10ABOT` architecture to complete a particular task is considered to be application software. The main purpose for creating a robotics framework is to allow

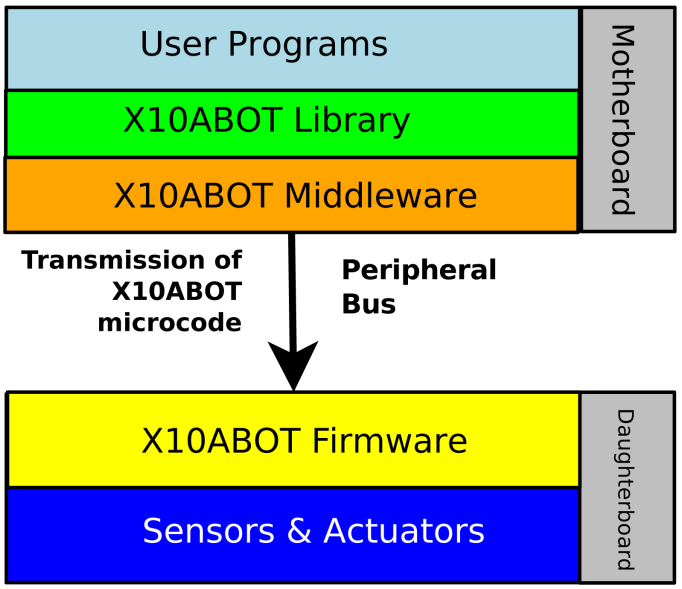

Figure 3.5. The `X10ABOT` software architecture

the end user to have a gratifying experience when developing software for their robotics project. We wanted to provide the users with the tools that would allow them to build more interesting robotics projects, without the usual limitations of scale or complexity that often follow.

We used Arduino's *Processing* language as our choice for the implementation of the framework. It is known to be very easy to learn and use and also has a supporting IDE that allows the user to get up and running in a short period of time. Applications will be written in a C-like syntax and compiled to run on an Arduino board. This is a familiar interface for many roboticists because Arduino is a very common robotics development platform.

3.1.2.3 The Daughterboard Firmware

The daughterboard firmware provides a standard interface from all daughterboards to their motherboard. All instructions to control or access sensors and actuators are communicated to it via the peripheral bus. These instructions are written as microcode abstractions of the hardware and need to be translated to actual assertions on the devices. Each abstracted command in the motherboard's `X10ABOT` Library is translated into one or more microcode hardware instructions. Microcode instructions are carried out sequentially by the daughterboards to manipulate the physical hardware components that are attached to their ports. The microcode instructions define a universal set of hardware operations. When these are put together, they can control most typical robotics sensors or actuators.

Our system was designed to support as many types of sensors and actuators as possible. To achieve this, we had to find a common denominator for all sensors and actuators. Sensors and actuators have many requisite hardware technologies that need to be supported. We realised that these technologies were subsets or a combination of a finite set of operations. Technologies such as pulse width modulation, hardware interrupts, digital input/output and analog input are just some of the basic ones used by most sensors and actuators. We determined that if we could define this set of hardware operations, we could control most if not all sensors and actuators. This would allow extensibility to currently existing and possibly new types of sensors and actuators.

The microcode instructions specified for daughterboards, provide different means of interacting with each port (see Table 3.1).

Commands	Parameters
digitalOut	state, db_address, port_number, pin_select
digitalIn	db_address, port_number, pin_select
pwm	pwm_select, db_address, port_number, duty_cycle
analog	db_address, port_number

Parameter Definitions

Format	Name	Description
byte	state	The state of the digital pin, HIGH(2), LOW(1), INPUT(0)
byte	db_address	The daughterboard address between 7 and 120, 0 for internal
byte	port_number	The daughter board's port number connected to the device
byte	pin_select	Choose which of the I/O pins on the port to use A(0) or B(1)
byte	pwm_select	Choose which of the PWM pins on the port to use A(0) or B(1)
byte	duty_cycle	Specify the duty cycle as a number between 0 and 255

Table 3.1: The complete set of hardware operations supported by the microcode interface to the daughterboards. These basic commands are the building blocks of all operations sent between the motherboard and daughterboard.

The firmware on the daughterboards will interpret these instructions and execute hardware procedures. These instructions are asserted to the standard sensor and actuator ports which are composed of pins, each with their specific purpose. Sensors and actuators are connected to their respective ports and respond to the assertions placed upon them by the daughterboard. Figure 3.6 shows a sequence diagram outlining the process from initializing an actuator in code to asserting a digital operation on the actuator.

The current microcode instructions include analog, digital and PWM and are the basic building blocks used to create more complex operations when combined or used with other computations. If a future sensor or actuator uses an incompatible operation, then a firmware upgrade which would include the new microcode instructions would have to be developed in order to interface with the new device (assuming that it could be made to be hardware compatible with the port). The microcode instructions are made to be very generic, ensuring compatibility with as many sensors and actuators as we could find at the time of design.

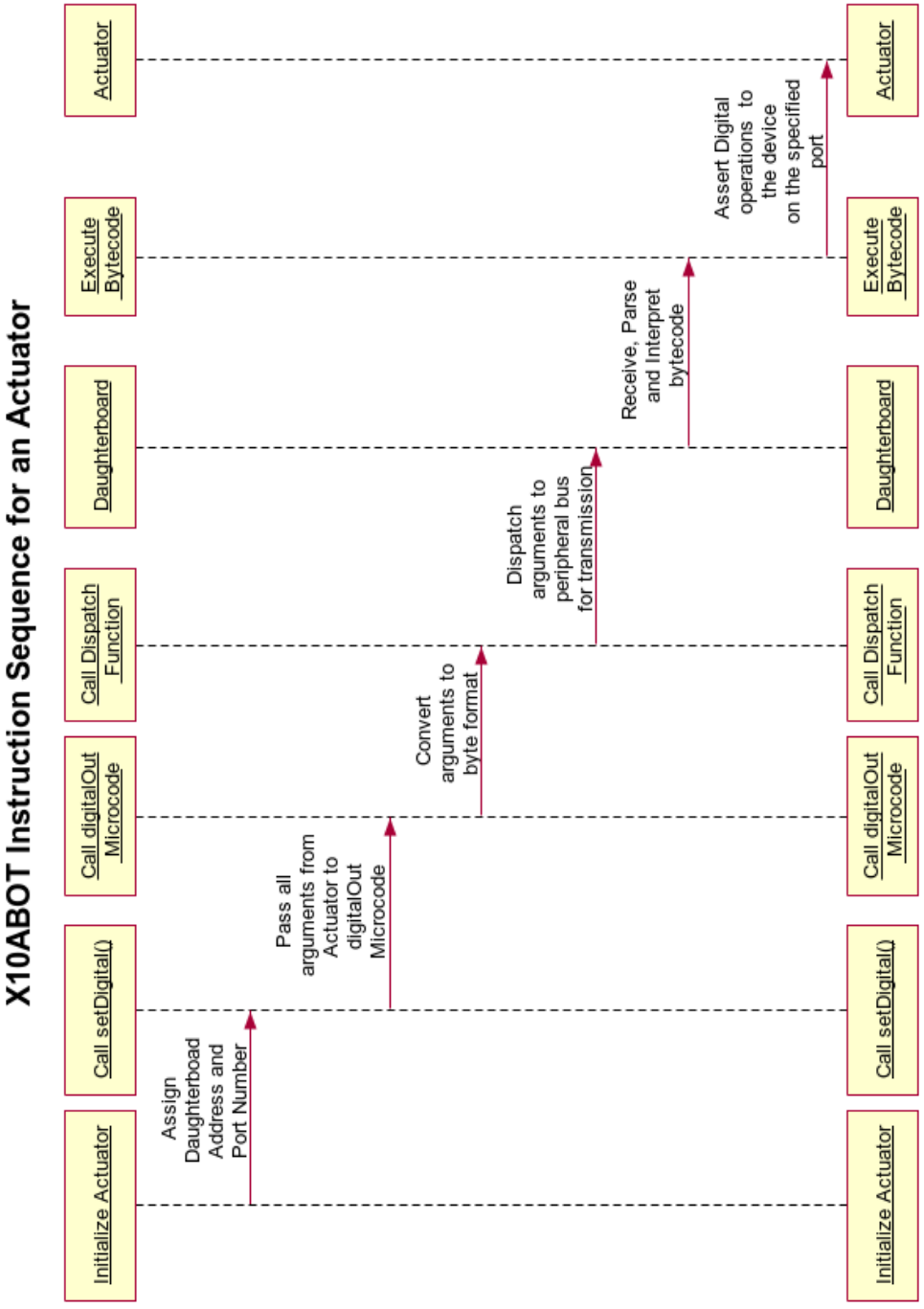

Figure 3.6. The components of the `X10ABOT` Architecture

The motherboard implements a number of functions that allow the libraries to utilize microcode instruction in the development of device libraries. These instructions are passed to the daughterboard where they are interpreted and executed. Figure 3.7 shows a few of the microcode instructions used in the `X10ABOT` library and their related parameters.

Microcode Implementation

```
/**
 * Asserts a state to a Digital IO pin
 *
 * @param  byte  state  The state of the digital pin, HIGH(2), LOW(1), INPUT(0)
 * @param  byte  db_address The daughterboard address between 7 and 120, 0 for motherboard
 * @param  byte  port_number the daughter board's port number connected to the device
 * @param  byte  pin_select choose which of the I/O pins on the port to use A(0) or B(1)
 * @return void
 */
void X10ABOT_MB::digitalOut(byte state, byte db_address,
                            byte port_number, byte pin_select){
  byte microcode[] =   {FN_IO+state,db_address,
                          ((port_number-1)<<1)+pin_select,incr_instr_seq()};
  dispatch(microcode, sizeof(microcode));
}
byte X10ABOT_MB::digitalIn(byte db_address, byte port_number, byte pin_select){
  byte seq_num = incr_instr_seq();
  byte microcode[] =   {FN_IO+IN,db_address,((port_number-1)<<1)+pin_select,seq_num};
  dispatch(microcode, sizeof(microcode));
   return requestHandler(microcode, 2,seq_num);
}
/**
 * Asserts a PWM signal to a pin
 *
 * @param  byte  pwm_select choose which of the PWM pins on the port to use A(0) or B(1)
 * @param  byte  db_address The daughterboard address between 7 and 120, 0 for motherboard
 * @param  byte  port_number the daughter board's port number connected to the device
 * @param  byte  duty_cycle specify the duty cycle as a number between 0 and 255
 * @return void
 */
void X10ABOT_MB::pwm(byte pwm_select, byte db_address,
                     byte port_number, byte duty_cycle){
  byte microcode[] =   {FN_PWM,db_address,((port_number-1)<<1)+pwm_select,
                        incr_instr_seq(),duty_cycle};
  dispatch(microcode, sizeof(microcode));
}

/**
 * Reads the state from an Analog pin
 *
 * @param  byte  db_address The daughterboard address between 7 and 120, 0 for motherboard
 * @param  byte  port_number the daughter board's port number connected to the device
 * @return int
 */
int X10ABOT_MB::analog(byte db_address, byte port_number){
  byte seq_num = incr_instr_seq();
  byte microcode[] =   {FN_ANALOG,db_address,((port_number-1)<<1),seq_num };
  dispatch(microcode, sizeof(microcode));
  return requestHandler(microcode, 6,seq_num);
}
```

Figure 3.7. Example implementation of the `X10ABOT` microcode instructions used in the device libraries

For every event on the daughterboard that is to be triggered, a microcode instruction is sent from the motherboard. The format of the instructions are as follows (see Table 3.2): The first four bits of the first byte define

the instructions to be executed and the next four bits define the specific operation for that instruction. The second byte specifies the daughterboard address, the third byte uses its leftmost bit to determine whether it is addressing a sensor or actuator port. The next six bits on the third byte form the port number and the rightmost bit is used for the pin selection. Each transmitted instruction has a sequence number that is assigned to each microcode instruction. Some instructions require additional data to complete their operations. The microcode format provides a sequence of data bytes that can be of arbitrary length. It was created to be used in operations where a lot of data may be transmitted e.g RS-232. Table 3.2 shows the structure of a typical microcode instruction.

Byte 1:	XXXX - - - -	FUNCTION BYTE
Byte 1:	- - - - XXXX	OPERAND BYTE
Byte 2:	XXXXXXXX	D.B. SELECTION
Byte 3:	X - - - - - - -	PORT TYPE
Byte 3:	- XXXXXX -	PORT SELECTION
Byte 3:	- - - - - - - X	PIN SELECTION
Byte 4:	XXXXXXXX	INSTRUCTION SEQUENCE NUMBER
Byte(s) 4+n	XXXXXXXX	DATA BYTES; n> 0

Table 3.2: Byte layout for transmitted microcode

In Figure 3.8 it can be further observed how these microcode instructions are used to create operations for the generic Actuator class. The code shows how microcode instructions are used to compose particular robotics instructions in the `X10ABOT` architecture. Here the I/O method accept HI and LO values according to the pins (**A and B**) on which the instruction will be asserted. The pwm instructions (**pwm_a and pwm_b**), named after the pwm pin which they assert, require the **power** parameter that is passed as an argument to the function.

The purpose of the daughterboard is to interpret these high-level instructions from the motherboard and execute them on the respective sensor or

Actuator Class Implementation

```
/*
* Sets digital pin values to High or Low
*/
void Actuator::setDigital(byte pin_a, byte pin_b){
  actuator.digitalOut(pin_a,_db,_port,A);
  actuator.digitalOut(pin_b,_db,_port,B);
}
/*
* Sets analog pin to a value between 0 and 100
*/
void Actuator::pwm_a(byte power){
  actuator.pwm(A, _db, _port, 255*power/100);
}
void Actuator::pwm_b(byte power){
  actuator.pwm(B, _db, _port, 255*power/100);
}
```

Figure 3.8. Example of the Actuator class methods that uses microcode to represent general operations on an actuator

actuator irrespective of the hardware specifics of the daughterboard. This ensures that high level language instructions sent from the motherboard remains platform independent.

3.1.2.4 Middleware

A middleware architecture is used to hide the low-level details of the communication interface between the components that make up a distributed robotics system. This level of abstraction allows for an interface between the main controller and the sensors and actuators which supports a high level definition of their operations.

The aspect of the `X10ABOT` architecture that transports the microcode from the libraries of the motherboard to the interpreter of the daughterboard is considered to be our middleware. Each microcode instruction formats the byte string as described in Table 3.2. The microcode then performs one or more of a series of dispatch and request events between the motherboard and the daughterboard. The daughterboard also initiates data transfers as well for certain special cases. All these operations are managed by the

Middleware Dispatch Function Implementation

```
/**
 * Dispatches microcode to the internal/external daughterboard for execution
 *
 * @param  byte*  microcode  This is an array of microcode instructions to be sent
 * @param  byte   byte_length the size in bytes of the microcode instruction
 * @return void
 */
void X10ABOT_MB::dispatch(byte* microcode, byte byte_length){
  if (microcode[D_B_SELECTION]==0){ //If 0 use internal Daughterboard
    db.localReceive(microcode, byte_length); //do operation locally
  }
  else{
    Wire.begin();   //Initiate TWI interface
    //Set TWI Transmission Speed
    TWBR=0x13; // (DEC = 19) ~300000L
    Wire.beginTransmission(microcode[D_B_SELECTION]); // transmit to device #x
    Wire.write(microcode, byte_length);       // sends all bytes
    i2cStatusLog(Wire.endTransmission()); // stop transmitting and log output
  }
  dispatchDataLog(microcode, sizeof(microcode));
}
```

Figure 3.9. Example of the Middleware Dispatch Function Implementation used to determine the transport interface and protocol of the operations between the motherboard and daughterboard.

middleware. The dispatch and request functions are used to pass data across the peripheral bus.

3.1.2.5 Extending the Library

To allow for easy extensibility and to support new hardware, the X10ABOT software architecture was designed to allow user extensible libraries. A library defines a set of instructions used to control a hardware device.

A new type of sensor or actuator can be supported by adding it as a device library on the X10ABOT motherboard. Creating a new device specification can be done without the need to understand any low level device specific coding, since all the configurations are abstracted for easier programming. Extensibility is therefore achieved through high-level abstraction of low level hardware components. Figure 3.11 shows the code for a new temperature sensor (thermistor) library that was created to extend the X10ABOT support for a new device. The thermistor required specialised manipulation

Middleware Request Handler Function Implementation

```
/**
 * Sends requests using microcode to the internal/external daughterboard for execution
 * @param  byte byte_length the size in bytes of the microcode instruction
 * @return void
 */
int X10ABOT_MB::requestHandler(byte* microcode, byte byte_length, byte seq_num){
  for (int i = 0; i < 3; ++i){
    if (_lookup[i][0]==seq_num){
      return _lookup[i][1];
    }
  }
  if (microcode[D_B_SELECTION]==0){
    byte return_array[6];
    db.localRequest(return_array);
    byte output[5];
    if (return_array[0]!=seq_num){
         //loop into cache
      _lookup[_lookup_index][0]= return_array[0];
      _lookup[_lookup_index][1]=return_array[1];
      if (_lookup_index>2){
        _lookup_index = 0;
      }else{
        _lookup_index++;
      }
    }else{
      for (int y=0; y<6;y++){
        output[y]=return_array[y+1];
      }
      int x = atoi((char*)output);
      return x;
    }
  }
  else{
    TWBR = 0x1B; // (DEC = 27) ~230000L
    Wire.requestFrom((int)microcode[D_B_SELECTION], 6);
    if(Wire.available()){    // slave may send less than requested
      byte c = Wire.read();
      if (c!=seq_num){
         //loop into cache
        _lookup[_lookup_index][0]= c;
        _lookup[_lookup_index][1]=Wire.read();
        if (_lookup_index>2){
            _lookup_index = 0;
          }else{
            _lookup_index++;
          }
        }else{
          byte result[5];
          byte x=0;
        while(Wire.available()){    // slave may send less than requested
          result[x]=Wire.read();
          x++;
        }
        int w = atoi((char*)result);
        return w;
      }
    }
    dispatchDataLog(microcode, sizeof(microcode));
  }
}
```

Figure 3.10. Example of the Middleware Request Handler Function Implementation used to determine the transport interface and protocol of the operations between the motherboard and daughterboard.

Actuator Class Implementation

```
#include "../X10ABOT_MB.h"
#include <math.h>
class Thermistor: public Sensor
{
  private:
    byte _db, _port;
  public:
    Thermistor(byte db_address, byte port_number):Sensor(_db, _port){
      _db = db_address;
      _port = port_number;
    }
    ~Thermistor(){};
    double readThermistorCelsius() {
        //The analog microcode requests the raw sensor value
        int RawADC = analog(_db,_port);
        double Temp;
        Temp = log(((10240000/RawADC) - 10000));
        Temp = 1/(0.001129148+(0.000234125+(0.0000000876741*Temp*Temp))*Temp);
        Temp = Temp - 273.15;            // Convert Kelvin to Celsius
        return Temp;
    }
    double readThermistorFarenheit() {
    // Convert Celsius to Fahrenheit
        return (readThermistorCelsius()* 9.0)/ 5.0 + 32.0;
    }
};
```

Figure 3.11. Thermistor library showing how a sensor can be supported through the extensibility features of the X10ABOT architecture

of the analog value it received from the sensor to produce a useful value in Celsius or Fahrenheit.

Both the hardware and software allow for inclusion of new and cutting edge technologies by utilising the simple idea of composition of low level microcode instructions. Many seemingly complex protocols or devices operate using only a few simple fundamental electronic operations defined by our microcode instructions. By creating a set of instructions that utilize these fundamental operations, any composite methods can be formed to complete most electronic tasks. For example: to operate an h-bridge, to control a DC motor, this requires a particular set of configurations of the digital I/O pins in their HI or LOW state, the configurations determine the operating mode: forward, reverse or stop. It may also require a pulse width modulation input to determine the motor's speed. These could be composed as a single library for the device. The X10ABOT system architecture was designed to facilitate device libraries which operate in a similar

sense as hardware drivers for computer peripherals. A driver typically communicates with the device through a communication subsystem to which the hardware connects.

3.2 The Arduino Implementation

`X10ABOT` is a robotics architecture that is indifferent to any particular hardware system. We endeavoured to create an architecture that had very few hardware specific restrictions that would force the developer to chose one hardware platform over another. We also tried to create an architecture that was capable of running on systems as simple as 8-bit micro-controllers. What we found was that a majority of hobbyist roboticists and robots used in school competitions tended to use 8-bit microcontrollers. However, many of the major notable robotics architectures were built on top of a full featured 32 or 64-bit processing systems. The architecture will however be able to scale up to a full scale 32 or 64-bit computer. This will allow for more complex programs to be written with more processor and hardware intensive operations. The most basic requirements for any implementation of the `X10ABOT` architecture is a microprocessor which has the ability to perform digital and analog input and output. In this case, one controller will operate as both motherboard and daughterboard. In order to support external daughterboards, each microcontroller must have support for the I^2C protocol in addition to the minimum requirements listed above.

The Arduino embedded system framework is a very active project, and it is notable for keeping at the cutting edge of embedded development. The framework is platform independent and currently deployed on a variety of

processors including Atmel's AVR 8-bit and 32-bit ARM processors. There are even a few other companies including Microchip who have adopted the Arduino framework for their 32-bit processors as well. The `X10ABOT` software was written with the Processing language supported by Arduino which is a derivative of C++. This allowed us to take advantage of many of C++'s high level language features including class hierarchy which is the basis of how development for the motherboard libraries are done. Arduino also comes with a cross platform IDE that allows development on all the major operating systems.

The Arduino platform provides very inexpensive hardware that can be used for both motherboard and daughterboard. They were designed for physical add-ons called shields which are convenient connection boards that can host auxiliary circuitry like port headers, external power and H-bridges. These shields are perfect platforms for the custom designed port of the `X10ABOT` daughterboards.

CHAPTER 4

RESULTS

This chapter presents evidence showing how the `X10ABOT` architecture achieves its objectives through the hardware and software platform. We demonstrate this by applying the framework across a number of practical examples before discussing the outcomes.

We demonstrate the framework in action by showing excerpts of sample programs for a simple mobile robot with a number of sensors and actuators distributed across one or more daughterboards. These examples demonstrate how modularity, scalability and extensibility are expressed through code and hardware configuration.

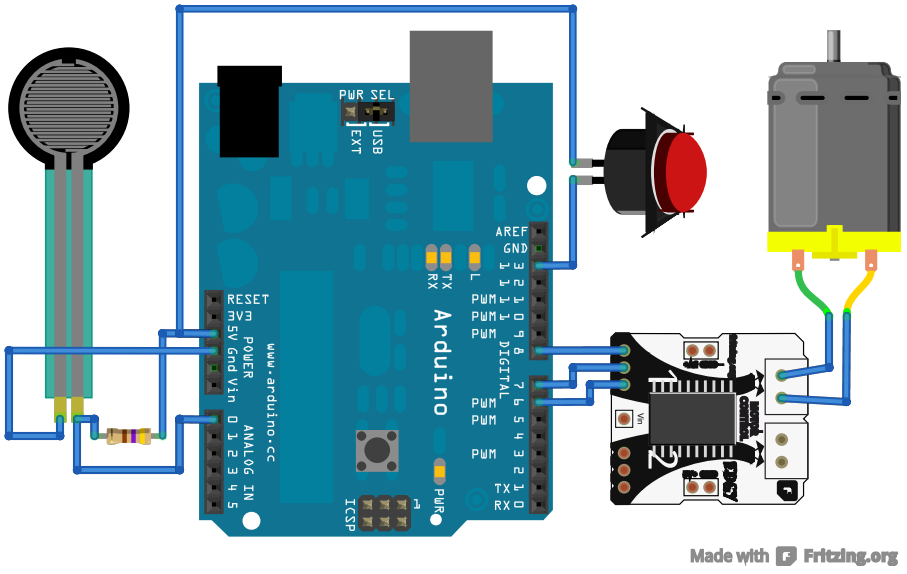

Figure 4.1. The components of a simple single-board `X10ABOT` setup

X10ABOT Sample code

```
/**
* Import the necessary libraries for the motherboard,
* internal daughterboard and the peripheral bus
**/
#include <Wire.h>
#include <X10ABOT_MB.h>
#include <X10ABOT_DB.h> //Include the internal daughterboard #0 (SELF)

//Initialise the DC motor on daughterboard #0 (SELF), actuator port #1
Actuator motor1(SELF,1);
//Initialise force sensor on daughterboard #0 (SELF), sensor port #1
Sensor force1(SELF,1);
//Initialise force sensor on daughterboard #0 (SELF), sensor port #2
Sensor pushbutton(SELF,1);

void setup(){}
void loop(){
//Continuously check the sensors for a reading
  if(pushbutton.readDigitalB() || (force1.readAnalog()>100)){
    motor1.setDigital(LO, HI); //Turn motor1 on by sending LO on pin a and HI on pin B
    motor1.pwm_a(100); //operate motor1 at full power (100%)
  }else{
    motor1.setDigital(LO, LO);//Turn motor1 off by sending LO on pin a and pin b
  }
}
```

Figure 4.2. Example of the X10ABOT architecture on a simple - single board robot

All tests and examples were done using the Arduino Prototyping system since this was the chosen platform used to prototype the X10ABOT architecture. The fundamental concepts however were general enough to be implemented on any modern microcontroller that supports I^2C and the operations listed in in Table 3.1.

Figure 4.2 is a typical code setup for a very simple mobile robot illustrated in Figure 4.1. All the sensors and actuators were placed on one board; for this particular implementation, we utilized the **internal** daughterboard which is hosted on the same board as the motherboard. This robot has one actuator (a DC motor connected to an external H-bridge), a force sensor and a push-button sensor.

Figure 4.2 is a simple program that drives the motor at full speed if either the touch sensor records an input or if there is a force on the force sensor above a certain threshold, otherwise the motor will be given the off signal.

In this instance, there are two types of devices that are declared, **Sensor** and **Actuator**. These represent the parent classes of all sensors and actuators respectively in the `X10ABOT` architecture. This example utilizes the parent class but for more complex sensors and actuators, a subclass would have been appropriate. All other sensor and actuator sub-types inherit from these two classes of devices, overriding where necessary.

It should be noted that it was necessary to include the library for both the motherboard and the daughterboard since they share the same Arduino board. Daughterboard address #0 or the constant **SELF** is reserved for the internal daughterboard on the same physical Arduino platform.

4.1 Modularity

Adding an extra capability was easily and seamlessly carried out using the `X10ABOT` architecture. We included an extra daughterboard that represented a complete subsystem using sensors and actuators. In our example in Figure 4.3, an extra DC motor and a light sensor were added as a single module. The `X10ABOT` architecture allowed for this new addition as a daughterboard via the peripheral bus. The module was just as easily added in software. The daughterboard was assigned a unique arbitrary address of **#9**. Instructions were then added to activate the new DC motor if there was light intensity above a certain threshold on the sensor. We could have easily controlled the existing motor or mixed the logic between the existing components, however for clarity we made it so that the modules would operate independently. In the Figure 4.4, we demonstrate how we added this extra functionality.

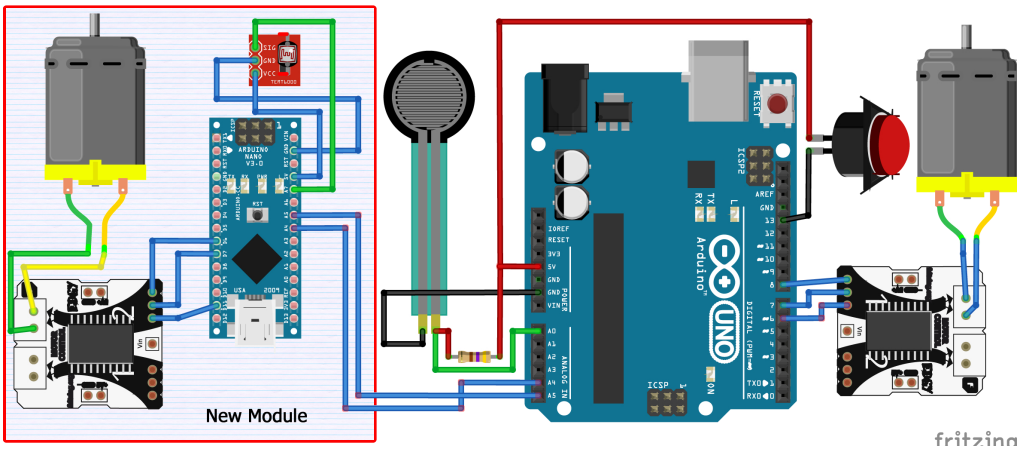

Figure 4.3. The components of a Modular single-board `X10ABOT` Example

Figure 4.4 shows, that even with the addition of two independent devices, *motor1* and *lightSensor* to the system, it did not affect the existing code but fit seamlessly in the development process. Separate code was added to carry out their operation which never needed to interact with the existing system. There needed to be no special accommodation for the new module in the existing code. There was no significant modification made to the existing hardware setup even with the fact that an extra physical component, another daughterboard, was added to the system. The `X10ABOT` design allowed a pluggable interface that facilitated adding new code and hardware with minimal modification to the existing setup. All these features are an indication of a truly modular architecture. As previously defined, we included an entire subsystem without modifying the existing components. The modules can be removed just as easily as they were installed. The level of expertise, with regard to specialised knowledge required to modify this system is also relatively minimal when compared to the same Arduino platform without the abstraction of the `X10ABOT` framework.

Modularity in Code

```
/**
* Import the necessary libraries for the motherboard,
* internal daughterboard and the peripheral bus
**/
#include <Wire.h>
#include <X10ABOT_MB.h>
#include <X10ABOT_DB.h> //Include the internal daughterboard #0 (SELF)

//Initialise the DC motor on daughterboard #0 (SELF), actuator port #1
Actuator motor1(SELF,1);
//Initialise force sensor on daughterboard #0 (SELF), sensor port #1
Sensor force1(SELF,1);
//Initialise force sensor on daughterboard #0 (SELF), sensor port #2
Sensor pushbutton(SELF,1);

void setup(){}
void loop(){
//Continuously check the sensors for a reading
if(pushbutton.readDigitalB() || (force1.readAnalog()>100)){
    motor1.setDigital(LO, HI); //Turn motor1 on by sending LO on pin a and HI on pin B
    motor1.pwm_a(100); //operate motor1 at full power (100%)
  }else{
    motor1.setDigital(LO, LO);//Turn motor1 off by sending LO on pin a and pin b
  }
```

```
 const byte BOARD2 = 9; //Constant with arbitrary daughterboard address
 //Declare motor on daughterboard #9, actuator port #1
 Actuator motor2(BOARD2,1);
 //Declare force sensor on daughterboard #9, sensor port #1
 Sensor lightSensor(BOARD2,1);

//Added extra functionality with a new sensor and a new actuator
//on daughterboard #9 while light is on the sensor, activate motor2
if (lightSensor.readAnalog()>700)
  {
    motor2.setDigital(HI, LO); //Turn motor2 on by sending HI on pin A and LO on pin b
    motor2.pwm_a(50); //operate motor2 at half power (50%)
  }else{
    motor2.setDigital(LO, LO);//Turn motor1 off by sending LO on pin a and pin b
  }
}
```

Figure 4.4. Example of code modularity (the code components were separated to emphasise modularity). Highlighted region indicates the code added for the new module

4.2 Extensibility and Abstraction

We demonstrate the X10ABOT architecture's ability to support new and varied types of sensors and actuators. This is accomplished by creating a subclass to either of the existing Sensor or Actuator classes. These parent classes support all the generic operations on all the available sensor and actuator ports. By creating a subclass, all the parent class properties and functions will be inherited. Devices with unique configurations can be supported by utilising the same abstracted daughterboard access available to the parent classes requiring no knowledge of the low-level details of the

Thermistor Library

```
#include "../X10ABOT_MB.h"
#include <math.h>
class Thermistor: public Sensor
{
  private:
    byte _db, _port;
  public:
    Thermistor(byte db_address, byte port_number):Sensor(_db, _port){
      _db = db_address;
      _port = port_number;
    }
    ~Thermistor(){};
    double readThermistorCelsius() {
        //The analog microcode requests the raw sensor value
        int RawADC = analog(_db,_port);
        double Temp;
        Temp = log(((10240000/RawADC) - 10000));
        Temp = 1/(0.001129148+(0.000234125+(0.0000000876741*Temp*Temp))*Temp);
        Temp = Temp - 273.15;            // Convert Kelvin to Celsius
        return Temp;
    }
    double readThermistorFarenheit() {
    // Convert Celsius to Fahrenheit
        return (readThermistorCelsius()* 9.0)/ 5.0 + 32.0;
    }
};
```

Figure 4.5. Example: A complete library for a thermistor temperature sensor. The highlighted region indicates where the X10ABOT microcode was added

hardware. In the following example, we will define a simple, yet complete library that reads and interprets the data from a thermistor temperature sensor. The raw analog value will be read then we will apply some computations to acquire a useful output. The class will support functions that return temperature values in both Celsius and Fahrenheit units. This code was originally sourced from the Arduino Playground (Malesevic and Stupic 2011) as a simple example to acquire thermistor temperature readings. We made a few simple modification that allowed the same code to be applicable to the X10ABOT architecture.

The conversion instructions of the X10ABOT library in Figure 4.5 is an almost exact replica of the fragment of code extracted from the Arduino Thermistor library. In the **readThermistorCelsius** function, the main change (highlighted in yellow) can be found where the code accesses the

Thermistor Example Application

```
#include <Wire.h>
#include "x10sions/Thermistor.h"
#include <X10ABOT_MB.h>
const byte FREEZER = 17; //Create a constant with the daughterboard address
const byte OVEN = 108;
const byte CONTROL_BOARD = 10;

//Declare thermistor on daughterboard #17, sensor port #1
Thermistor coldSensor(FREEZER,1);
//Declare thermistor on daughterboard #108, sensor port #6
Thermistor hotSensor(OVEN,6);
//Declare digital alarm on daughterboard #10, actuator port #3
Actuator alarm(CONTROL_BOARD,3,A);

void setup(){}
void loop(){
//Continuously check the sensors to see if temperature within threshold
  if((coldSensor.readThermistorCelsius() > 1 )
       || hotSensor.readThermistorFarenheit()<200){
  alarm.on_a(); //Turn on alarm
  }
}
```

Figure 4.6. Example application of the thermistor temperature sensor library

analog pin to read the raw data value acquired from the sensor. These functions are at the lowest level of abstraction and are used to read and write to the individual pins of each port. They include *digitalIn, digitalOut, pwm and analog.* For this particular case, the **analog(__db,__port)** microcode function was invoked, access to this function was inherited by extending the Sensor class. This is a method used to abstract sensor port's analog pin so that it becomes hardware independent. The sensor port (**__port**) belongs on a daughterboard (**__db**)where it accesses the data from the analog pin. The raw analog value is then returned to the calling function where it is further processed.

This thermistor library can then be easily utilized by the user as follows (See Figure 4.6):

The code in Figure 4.6 presents a temperature monitoring system that watches the temperature that it receives from two thermistors. If the value read from the thermistors goes beyond particular thresholds for either of them, an alarm is triggered.

A typical end user would not be involved in creating libraries for sensors. They would have only been exposed to the setup in Figure 4.6. The only requisite knowledge would be familiarity with the supported list of functions specified for a thermistor. In this case only **readThermistorFarenheit()** and **readThermistorCelsius()** are specified and both have been used. The library for a thermistor was included and it became very straightforward to utilize the sensor readings afterwards.

4.3 Scalability

Based on the results gathered from testing for modularity, an inherent property for scalability can be observed. The method of modular addition lends itself to be scalable to tens of modules as defined by the specifications of the I^2C peripheral bus protocol. Scalability is not only measured by how many devices can be connected to the system, but we must take into consideration the performance of the system when all these devices are operational. To investigate this, we benchmarked the performance of the architecture that we implemented on the Arduino platform. When operating with daughterboards across the peripheral bus, the `X10ABOT` architecture's major limiting factor becomes the speed of data transfer. The default speed of the I^2C peripheral bus when using the Arduino platform is 100Kb/s, this can be further configured to go as high as 400Kb/s. For our application on the Arduino we had best result when performing between 230Kb/s and 300Kb/s. We performed a number of tests to determine the latency of each type of basic instruction (microcode) to see how they would affect the performance of the entire system. All tests below were carried out on the Arduino Mega, using the Atmel ™ATMega1280 microcontroller.

We carried out instruction execution latency tests to evaluate the performance of the `X10ABOT` framework on the Arduino platform. We performed three sets of test comparing the speed of execution on:

- An Arduino without the `X10ABOT` framework.
- The `X10ABOT` framework with a single daughterboard on the peripheral bus

- The `X10ABOT` framework with no external daughterboards (internal daughterboard used instead).

The tests were carried out by invoking and storing the value of Arduinoś **micros()** function which returns the number of microseconds since the program started. The times before and after function executions were recorded and their difference taken. The timing utility on the Arduino has a maximum resolution of 4 micro second intervals therefore the readings were not precise. We implemented the tests in a continuous loop and recorded a sample of 20 consecutive duration time values for each tested function. We did further computations to find the mean and the standard deviation of the sample set of these values.

The tables below display the result of our timing experiment:

Table 4.1: Table showing instruction latency in microseconds(μs) for a regular Arduino using a sample of 20 consecutive executions

Direct Arduino Board readings				
	digitalOut	digitalIn	pwm	analog
Mean:	13.6	15.6	12.2	115.6
Standard Deviation:	2.01	2.21	0.89	3.41
3 Stnd. Devs. Lower Limit	7.57	8.97	9.53	105.37
3 Stnd. Devs. Upper Limit	19.63	22.23	14.87	125.83

Table 4.2: Table showing microcode instruction latency in microseconds(μs) for the on-board daughterboard using a sample of 20 consecutive executions

One Board `X10ABOT` readings				
	digitalOut	digitalIn	pwm	analog
Mean:	39.4	83.58	57.8	215.58
Standard Deviation:	2.15	2.27	4.17	2.95
3 Stnd. Devs. Lower Limit	32.95	76.77	45.29	206.73
3 Stnd. Devs. Upper Limit	45.85	90.39	70.31	224.43

Table 4.3: Table showing microcode latency in microseconds(μs) for a daughterboard connected over the peripheral bus using a sample of 20 consecutive executions

Two Board `X10ABOT` readings				
	digitalOut	digitalIn	pwm	analog
Mean:	261.4	554.8	319	685.2
Standard Deviation:	2.98	2.63	2.20	5.43
3 Stnd. Devs. Lower Limit	252.46	546.91	312.4	668.91
3 Stnd. Devs. Upper Limit	270.34	562.69	325.6	701.49

4.4 Observations

From the results, we have observed that the latency for output operations took notably less time than that of input operations. This was expected since input operations consist of two instructions over the peripheral bus.

The first is a request for the data and the second is a retrieval. We had to intentionally slow down the speed of the second request from 300Kb/s to 230Kb/s to allow time for the daughterboard to retrieve the requested data. We attempted faster speeds but it resulted in periodic data loss. Output operations only required that one instruction be sent across the bus.

We calculated the standard deviation and presented the data ranges for up to three standard deviations. These upper and lower limits indicate the statistical prediction of the maximum and minimum execution time for each function with about 99.7 percent confidence. The end user can then determine if this latency is acceptable for their application.

We also presented a comparative analysis of the Arduino with and without the `X10ABOT` system. We observed that the all readings on all platforms fell below the 1 millisecond mark. For most school based or hobbyist projects, this falls within an acceptable range.

The results emphasised that the advantages of `X10ABOT` platform are more pronounced when it has to manipulate devices across multiple daughterboards. There are trade-offs in performance which had to be sacrificed to accomplish this but these trade-offs allowed the framework to achieve its goals of being modular, scalable and extensible.

CHAPTER 5
CONCLUSION AND DISCUSSION

By actualising the architectural design concepts of modularity, scalability and extensibility discussed in Chapter 2. We were able to create a platform that effectively accomplished these required objectives. We have also realised that this design framework has created an extensible platform for numerous avenues of future work. There are opportunities to support various programming interfaces, porting `X10ABOT` to more powerful hardware and support of additional sensor and actuator devices.

5.1 Limitations

The efficiency of the `X10ABOT` architecture is limited by the speed of its peripheral bus, which in our case is I^2C and the power of the processors that implement it. The limitation details are listed below:

- Our implementation was done on an 8-bit Arduino processor running at 16MHz with 8KB of SRAM. This inherently limited the kinds of tasks that could have been attempted. This is not an insurmountable obstacle however for most projects within the scope of the target user's applications, this would be adequate. There have also been recent advancement in the Arduino community where there is now available 32-bit Arduinos running at 84Mhz with 96KB of SRAM. Using this hardware could increase the speed of operations on the

X10ABOT, possibly positively impacting the data in the results section of this thesis without significantly increasing the overall cost. This would be mostly due to the ability of faster processing of instructions. Porting the existing code base to this platform would require very minor changes since Arduino code is mostly hardware independent across other Arduino compatible products.

- Communication between motherboard and daughterboard are limited by the speed on the I^2C bus which is not only limited by its specification but the clock speed of the host processor. As mentioned previously, better hardware would overcome this limitation. Applications that may require a more powerful system may include ones with significant amounts of video processing.

- Other limitations include the I^2C 112 device limit on the peripheral bus, and the limit of 16 sensors and actuators (8 each) per daughterboard. These limits are relatively very high and are more than adequate for applications within the scope of this project.

- To achieve the goals of modularity, scalability and extensibility, we made a design decision to create specialised input and output ports. These ports are further specialised by having pins dedicated to certain functionalities. This decision limited the number of sensors and actuators that can be attached to a daughterboard. We compared the number of ports available when utilising the X10ABOT framework on a single Arduino Mega (4 sensor ports, 6 actuator ports) to the number provided by a popular kit like the LEGO Mindstorms NXT (4 sensor ports, 3 actuator ports). When we compared for cost, the Arduino Mega cost approximately $60 and the LEGO Mindstorms

NXT approximately $260 US dollars. Based on these results, the Arduino with the X10ABOT framework provided a lower cost per port value. The current implementation of the X10ABOT framework may not be as feature complete as the LEGO Mindstorms NXT but for basic sensors and actuators operations, it provides a less expensive way to have more.

5.2 Achievements

The following are a list of the main objectives that were accomplished:

- **Designed modular, distributed hardware framework:** We are now able to easily add or remove hardware and software modules in and out out of the system.
- **Engineered a model that can support almost 900 sensors and 900 actuators:**Using the theoretic capabilities of the I^2C peripheral bus, we are able to support up to 112 daughterboards supporting 8 sensor and 8 actuators each.
- **Created an extensible software framework that can support the addition of many types of sensors & actuators:** Taking advantage of the abstraction of ports and pins accomplished in the framework design, we were able to create a platform for supporting the creation of libraries that support new sensors and actuators.
- **Created a flexible system that allows easy interoperability with distinct hardware architectures:** The X10ABOT was designed with a middle-ware type software layer that allows the system

to communicate with hardware platforms that are very different from each other, providing that both are using the standard protocol.

- **Kept the cost low by using the Arduino as the main hardware component:** The system was designed to be implemented on very inexpensive hardware such as Arduino based devices. It can scale to support a large number of these inexpensive devices.

- **Successfully tested the framework with both digital and analog sensors and actuators for latency:** We benchmarked the system to determine its suitability for robotics applications which student's and hobbyists may pursue. We found the performance to be acceptable and where more power is required, more powerful hardware can be acquired.

We can therefore conclude that the `X10ABOT` architecture provides a suitable platform that will enable efficient robotics development for our target audience of students and hobbyists. It will accomplish this by providing a model that can scale to a large number of sensors and actuators. This was made possible by utilising a distributed modular framework of daughterboards that can be attached whenever needed. We performed tests to measure the latency of I/O instructions between the peripheral devices and the motherboard. We found it to be in an acceptable range that was tolerable for most robotics applications within the scope of 8-bit microprocessors. This delay time would also see improvements if the processor and communication speeds were increased. The performance of the system is dependent upon the hardware that implements it.

5.3 Future Work

In order to facilitate processor intensive operations, the architecture would have to be ported to a more powerful hardware platform. At the time of writing, there was a recent release of an Arduino based on the 32-bit, 84Mhz Arm core processor with 96 Kbytes of SRAM. There is also the possibility of porting the architecture over to the Raspberry Pi for an even more powerful motherboard at about half the price of an Arduino. The Raspberry Pi was not yet developed at the time this project began.

The code now has support for basic analog and digital sensors. Future work could involve expanding the core offering for more advanced and popular sensors and actuators and the creation of an online library where a community could contribute libraries for their favourite sensors and actuators.

Currently the only supported method to write program code is using the Arduino framework. We had begun work on a bytecode interpreter that would allow the `X10ABOT` framework to read in bytecode from the LEGO Mindstorms' broad range of programming interfaces. Future work could see the completion of this interpreter that would provide a useful tool for roboticists with Mindstorms experience.

The `X10ABOT` platform has a lot of possibilities with respect to its usage as a part of the new Internet of Things (IOT) trend. Its ability to have numerous input and output interfaces can facilitate remote monitoring and activity via an internet connection on the motherboard.

REFERENCES

Avcı, Akın. 2008. “The Universal Robot Bus: A Local Communication Infrastructure for Small Robots.” PhD diss., Bilkent University.

Bakken, David E. 2001. “Middleware.” *Encyclopedia of Distributed Computing.*

Barker, Bradley S, and John Ansorge. 2007. “Robotics as Means to Increase Achievement Scores in an Informal Learning Environment.” *Journal of Research on Technology in Education* 39 (3): 229–243.

Benitti, Fabiane Barreto Vavassori. 2012. “Exploring the Educational Potential of Robotics in Schools: A Systematic Review.” *Computers & Education* 58 (3): 978–988. doi:https://doi.org/10.1016/j.compedu.2011.10.006.

Bonarini, Andrea, Matteo Matteucci, Martino Migliavacca, and Davide Rizzi. 2012. “R2P: an Open Source Modular Architecture for Rapid Prototyping of Robotics Applications.” 1st IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control, *IFAC Proceedings Volumes* 45 (4): 68–73. doi:http://dx.doi.org/10.3182/20120403-3-DE-3010.00051.

Bonarini, Andrea, Matteo Matteucci, Martino Migliavacca, Roberto Sannino, and Daniele Caltabiano. 2011. “Modular Low-Cost Robotics: What Communication Infrastructure?” 18th IFAC World Congress, *IFAC Proceedings Volumes* 44 (1): 917–922. doi:http://dx.doi.org/10.3182/20110828-6-IT-1002.03274.

Elkady, Ayssam, and Tarek Sobh. 2012. “Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography.” *Journal of Robotics* 2012:1–15. doi:10.1155/2012/959013.

Enderle, Stefan, Stefan Sablatnög, Steffen Simon, and Gerhard K Kraetzschmar. 2000. “Tetrixx–A Robot Development Kit.” *Proc. of Edutainment Robots WS:* 1–3.

Ferrell, Cynthia L. 1995. “Global behavior via Cooperative Local Control.” *Autonomous Robots* 02139 (2): 105–125. doi:10.1007/BF00735430.

Gates, Bill. 2007. “A Robot in Every Home.” *Scientific American* 296, no. 1 (January): 58–65. doi:10.1038/scientificamerican0208-4sp.

Gerkey, B, and K Conley. 2011. "Robot Developer Kits [ROS Topics]." *Robotics Automation Magazine, IEEE* 18 (3): 16. doi:10.1109/MRA.2011.942483.

Himpe, Vince. 2015. "I2C (Inter-Integrated Circuit) Bus Technical Overview & Frequently Asked Questions." Accessed June 12, 2015. http://www.esacademy.com/en/library/technical-articles-and-documents/miscellaneous/i2c-bus.html.

Kramer, James, and Matthias Scheutz. 2006. "Development Environments for Autonomous Mobile Robots: A Survey." *Autonomous Robots* 22, no. 2 (December): 101–132. doi:10.1007/s10514-006-9013-8.

Malesevic, Milan, and Zoran Stupic. 2011. "Thermistor Example." Accessed September 17, 2011. http://playground.arduino.cc/ComponentLib/Thermistor2.

Martin, Fred G. 2000. "The Handy Board Technical Reference." Accessed June 22, 2015. http://akbar.marlboro.edu/~mahoney/handyboard/misc_docs/hbmanual.pdf.

Mirats Tur, J M, and Carlos F Pfeiffer. 2006. "Mobile Robot Design in Education." *Robotics Automation Magazine, IEEE* 13, no. 1 (March): 69–75. doi:10.1109/MRA.2006.1598055.

Oraw, B, and J Tinder. 2004. "Design Report MARS: Modular Autonomous Robotic Snake." *Relation* 10 (1.51): 8903.

Semiconductors, N X P. 2007. "I2C-bus Specification and User Manual." Accessed May 16, 2015. http://www.nxp.com/documents/user_manual/UM10204.pdf.

Utz, H, et al. 2005. "Advanced Software Concepts and Technologies for Autonomous Mobile Robotics." PhD diss., University of Ulm. doi:10.18725/oparu-350.

Ventura, Rodrigo, Pedro Aparício, Carlos Marques, Pedro Lima, and Luís Custódio. 2000. "ISocRob — Intelligent Society of Robots." In *RoboCup-99: Robot Soccer World Cup III,* edited by Manuela Veloso, Enrico Pagello, and Hiroaki Kitano, 1856:715–718. Lecture Notes in Computer Science. Springer Berlin Heidelberg. doi:10.1007/3-540-45327-X_89.

Weiss, Richard, and Isaac Overcast. 2008. "Finding your Bot-mate: Criteria for Evaluating Robot Kits for Use in Undergraduate Computer Science Education." *J. Comput. Sci. Coll.* (USA) 24, no. 2 (December): 43–49.

Yang, Xiaoli Yang Xiaoli, Yaqi Zhao Yaqi Zhao, Wei Wu Wei Wu, and Hui Wang Hui Wang. 2008. "Virtual Reality Based Robotics Learning System." In *2008 IEEE International Conference on Automation and Logistics,* 859–864. September. Ieee. doi:10.1109/ICAL.2008.4636270.

APPENDIX I - X10ABOT SOURCE CODE

The following code consist of all the files involvd in the `X10ABOT` software framework.

x10abot/X10ABOT_MB.h

```
/*
  X10ABOT_MB.h - Library for flashing X10ABOT_MB code.
  Created by Rohan A. Smith, January 31, 2012.
  Released into the public domain.
*/


#ifndef X10ABOT_MB_H
#define X10ABOT_MB_H

#define LOGGING 0
#include <Wire.h>
#include "Arduino.h" //"WProgram.h"
//#include "Actuator.h"


//Functions
static const byte FN_IO = 1 << 4;
static const byte FN_PWM  = 2 << 4;
static const byte FN_ANALOG = 3 << 4;
static const byte FN_SERIAL = 4 << 4;


//IO Operands
static const byte OP_IO_PI_LOW = 4;  //pulse in falling edge
static const byte OP_IO_PI_HI = 3;  //pulse in rising edge
static const byte OP_IO_HI = 2;
static const byte OP_IO_LOW = 1;
static const byte OP_IO_INP = 0;

//IO Operands
static const byte HI = 2;
static const byte LO = 1;
static const byte IN = 0;

//NULL OPERATION
static const byte OP_NOP = 0;

//Ports 1 - 4
static const byte PORT_1 = 0 << 1;
```

```
41  static const byte PORT_2 = 1 << 1;
42  static const byte PORT_3 = 2 << 1;
43  static const byte PORT_4 = 3 << 1;
44
45  //Port Types
46  static const byte SENSOR= 64;
47  static const byte ACTUATOR = 0;
48
49  //Pins
50  static const byte PIN_A = 0;
51  static const byte PIN_B = 1;
52  static const byte A = 0;
53  static const byte B = 1;
54
55  //Microcode Array
56  static const byte FUNCTION_OPERAND = 0;
57  static const byte D_B_SELECTION = 1;
58  static const byte PORT_PIN = 2;
59  static const byte SEQ_NUM = 3;
60  static const byte INSTR_HEADER_SIZE = 4;
61
62
63  class X10ABOT_MB{
64
65    public:
66      X10ABOT_MB(byte logging);
67      ~X10ABOT_MB();
68
69      void dispatch(byte* pattern, byte byte_length);
70      int requestHandler(byte* microcode, byte byte_length, byte seq_num);
71      void test_function();
72
73      //fundamental microcode operations
74      void digitalOut(byte state, byte db_address, byte port_number, byte
          ↪  operation);
75      byte digitalIn(byte db_address, byte port_number, byte operation);
76      void pwm(byte pwm_select, byte db_address, byte port_number, byte
          ↪  duty_cycle);
77      int analog(byte db_address, byte port_number);
78      /**
79      * Logging Functions
80      **/
81      void i2cStatusLog(byte var);
82      void dispatchDataLog(byte * dispatch, int size);
83      byte incr_instr_seq();
84
85    private:
86      int _logging;
87      int _instr_seq;
88      byte test_pattern[3];
89      byte test_pattern2[3];
90      byte _lookup[3][2];
91      byte _lookup_index;
```

```
92      int _analog, _digital;
93
94  };
95
96  #endif
97
98  #ifndef ACTUATOR_H
99  #define ACTUATOR_H
100
101 //#include "X10ABOT_MB.h"
102
103 class Actuator: public X10ABOT_MB
104 {
105
106   public:
107     Actuator(byte db_address, byte port_number):X10ABOT_MB(LOGGING){
108       _db = db_address;
109       _port = port_number+ACTUATOR;
110     }
111     Actuator(byte db_address, byte port_number, byte
        ↪  pin_select):X10ABOT_MB(LOGGING){
112       _db = db_address;
113       _port = port_number+ACTUATOR;
114       _pin = pin_select;
115     }
116     //Actuator(byte db_address, byte port_number, byte pin_select);
117     ~Actuator();
118
119     void run(byte power);
120     void stop();
121     void on(byte power);
122     void on();
123     void on_a();
124     void on_b();
125     void pwm_a(byte power);
126     void pwm_b(byte power);
127     void off();
128     void off_a();
129     void off_b();
130
131   private:
132     byte _db, _port, _pin;
133     byte _lookup[3][2];
134 };
135
136 #endif
137
138
139 #ifndef SENSOR_H
140 #define SENSOR_H
141
142 //#include "X10ABOT_MB.h"
143
```

```
144 class Sensor: public X10ABOT_MB
145 {
146
147   public:
148     Sensor(byte db_address, byte port_number):X10ABOT_MB(LOGGING){
149       _db = db_address;
150       _port = port_number+SENSOR;
151     }
152     Sensor(byte db_address, byte port_number, byte
        ↪  pin_select):X10ABOT_MB(LOGGING){
153       _db = db_address;
154       _port = port_number+SENSOR;
155       _pin = pin_select;
156     }
157     //Sensor(byte db_address, byte port_number, byte pin_select);
158     ~Sensor();
159     byte readDigital();
160     void writeDigitalLo();
161     void writeDigitalHi();
162     void readDigitalA();
163     void readDigitalB();
164     byte readDigital(byte power);
165     int readAnalog();
166     int getAnalog();
167     void off();
168
169   private:
170     byte _db, _port,_pin;
171     int _analog, _digital;
172 };
173
174 #endif
```

x10abot/X10ABOT_MB.cpp

```
1  /**
2   * X10ABOT_MB.h - Library for flashing X10ABOT_MB code.
3   * Created by Rohan A. Smith, January 31, 2012.
4   * Released into the public domain.
5   *
6   * Long description for class (if any)...
7   *
8   * @copyright  2013 Rohan Smith
9   * @license    http://www.zend.com/license/3_0.txt   PHP License 3.0
10  * @version    Release: @package_version@
11  * @link       http://dev.zend.com/package/PackageName
12  * @since      Class available since Release 1.2.0
13  */
14
15 /*
16   X10ABOT_MB.h - Library for flashing X10ABOT_MB code.
17   Created by Rohan A. Smith, January 31, 2012.
18   Released into the public domain.
```

```
19  */
20
21  #include "X10ABOT_MB.h" //include the declaration for this class
22  #include <X10ABOT_DB.h> //include the declaration for this class
23  #include <Wire.h>
24    //#include <MemoryFree.h>
25
26  X10ABOT_DB db(LOGGING);
27
28
29  /**
30   * X10ABOT_MB Constructor
31   *
32   * @param  byte logging  Toggles logging over Serial ON(1) OFF(0)
33  */
34  X10ABOT_MB::X10ABOT_MB(byte logging){
35    _logging = logging;
36    _instr_seq = 0;
37    _lookup_index = 0;
38  }
39
40  /**
41   * X10ABOT_MB Destructor
42   * no params
43   */
44  X10ABOT_MB::~X10ABOT_MB(){
45    /*nothing to destruct*/
46  }
47
48  /**
49   * X10ABOT_MB Destructor
50   * no params
51   */
52  byte X10ABOT_MB::incr_instr_seq(){
53    if (_instr_seq>250)
54    {
55      _instr_seq = 0;
56    }else{
57      _instr_seq = _instr_seq+1;
58    }
59    return _instr_seq;
60  }
61
62  /**
63   * Dispatches microcode to the internal/external daughterboard for
    ↪  execution
64   *
65   * @param  byte*  microcode  This is an array of microcode instructions
    ↪  to be sent
66   * @param  byte   byte_length the size in bytes of the microcode
    ↪  instruction
67   * @return void
68   */
```

```
69  void X10ABOT_MB::dispatch(byte* microcode, byte byte_length){
70    if (microcode[D_B_SELECTION]==0){
71      db.localReceive(microcode, byte_length);
72    }
73    else{
74     Wire.begin();
75     // Read and write here to 400kHz devices
76     //((16000000MHz/400000L)-16)/2 =
77     //TWBR=0x0C; // (DEC = 12) 400000L
78     TWBR=0x13; // (DEC = 19) ~300000L
79     //TWBR = 0x20; // (DEC = 32) 200000L
80     //TWBR=0x48; // (DEC = 72) DEFAULT 100000L
81
82      Wire.beginTransmission(microcode[D_B_SELECTION]); // transmit to
         ↪  device #x
83      Wire.write(microcode, byte_length);              // sends all bytes
84      i2cStatusLog(Wire.endTransmission());    // stop transmitting
85    }
86    dispatchDataLog(microcode, sizeof(microcode));
87  }
88
89  /**
90   * Sends requests using  microcode to the internal/external daughterboard
     ↪  for execution
91   *
92   * @param  byte   byte_length the size in bytes of the microcode
     ↪  instruction
93   * @return void
94   */
95  int X10ABOT_MB::requestHandler(byte* microcode, byte byte_length, byte
     ↪  seq_num){
96    for (int i = 0; i < 3; ++i){
97      if (_lookup[i][0]==seq_num){
98        return _lookup[i][1];
99      }
100
101   }
102   if (microcode[D_B_SELECTION]==0){
103     byte return_array[6];
104     db.localRequest(return_array);
105     //Serial.print("return_array0: ");Serial.println(return_array[0]);
106     //Serial.print("return_array1: ");Serial.println(return_array[1]);
107     //Serial.print("seq#: ");Serial.println(seq_num);
108
109     //_lookup[_lookup_index][0]=return_array[0];
110     //_lookup[_lookup_index][1]=return_array[1];
111     byte output[5];
112     if (return_array[0]!=seq_num){
113
114          //loop into cache
115       _lookup[_lookup_index][0]= return_array[0];
116         //for (int j = 0; j < 1; j++){
117       _lookup[_lookup_index][1]=return_array[1];
```

```
118         if (_lookup_index>2){
119           _lookup_index = 0;
120         }else{
121           _lookup_index++;
122         }
123       }else{
124 
125         for (int y=0; y<6;y++){
126           output[y]=return_array[y+1];
127         }//Serial.println(freeMemory());Serial.println(' ');
128         int x = atoi((char*)output);
129         return x;
130       }
131     }
132     else{
133       //Wire.begin();
134       //Wire.beginTransmission(microcode[D_B_SELECTION]); // transmit to
          ↪  device #x
135       //Wire.write(microcode, byte_length);              // sends all bytes
136       //i2cStatusLog(Wire.endTransmission());    // stop transmitting
137        // Read and write here to 400kHz devices
138      //((16000000MHz/400000L)-16)/2 =
139      //TWBR=0x0C; // (DEC = 12) 400000L
140      //TWBR=0x13; // (DEC = 19) ~300000L
141       TWBR = 0x1B; // (DEC = 27) ~230000L
142      //TWBR = 0x20; // (DEC = 32) 200000L
143      //TWBR=0x48; // (DEC = 72) DEFAULT 100000L
144 
145       Wire.requestFrom((int)microcode[D_B_SELECTION], 6);
146       if(Wire.available()){    // slave may send less than requested
147         byte c = Wire.read();
148         if (c!=seq_num){
149           //Serial.print(c);
150           //Serial.print(" UNEQUAL TO ");
151           //Serial.print(seq_num);
152            //loop into cache
153           _lookup[_lookup_index][0]= c;
154           //for (int j = 0; j < 1; j++){
155           _lookup[_lookup_index][1]=Wire.read();
156           if (_lookup_index>2){
157               _lookup_index = 0;
158             }else{
159               _lookup_index++;
160             }
161           }else{
162           //DO SOMETHING WITH c
163             byte result[5];
164             byte x=0;
165           while(Wire.available()){    // slave may send less than requested
166             result[x]=Wire.read();
167 //Serial.print("Result: ");Serial.println(result[x]);
168             x++;
169           }
```

```
170           int w = atoi((char*)result);
171           //Serial.print("Source SensorData X: ");Serial.println(w);
172           return w;
173         }
174       }
175
176       dispatchDataLog(microcode, sizeof(microcode));
177     }
178   }
```

x10abot/X10ABOT_DB.h

```
1   /*
2     X10ABOT_DB.h - Library for flashing X10ABOT_DB code.
3     Created by Rohan A. Smith, January 31, 2012.
4     Released into the public domain.
5   */
6
7
8   #ifndef X10ABOT_DB_H
9   #define X10ABOT_DB_H
10
11
12  #include "Arduino.h" //"WProgram.h"
13  //#include <Wire.h>
14
15  static const byte PORT_A = 1;
16  static const byte PORT_B = 2;
17  static const byte PORT_C = 3;
18  static const byte PORT_D = 4;
19  static const byte SELF = 0;
20
21  //Functions
22  static const byte DB_FN_IO = 1;
23  static const byte DB_FN_PWM  = 2;
24  static const byte DB_FN_ANALOG = 3;
25  static const byte DB_FN_SERIAL = 4;
26
27  //IO Operands
28  static const byte DB_OP_IO_PI_LOW = 4;  //pulse in falling edge
29  static const byte DB_OP_IO_PI_HI = 3;  //pulse in rising edge
30  static const byte DB_OP_IO_HI = 2;
31  static const byte DB_OP_IO_LOW = 1;
32  static const byte DB_OP_IO_INP = 0;
33
34  //PWM Operands
35  //static const byte OP_PWM_A = 0;
36  //static const byte OP_PWM_B = 1;
37
38  //Microcode Array
39  static const byte DB_FUNCTION_OPERAND = 0;
40  static const byte DB_D_B_SELECTION = 1;
41  static const byte DB_PORT_PIN = 2;
```

```
42  static const byte DB_SEQ_NUM = 3;
43  static const byte DB_INSTR_HEADER_SIZE = 4;
44
45  //port to pin assignment
46
47  typedef struct {
48        byte io_pin[2];  //Input/Output pins
49        byte analog; //Analog input pin
50      }INPUT_PORT;
51
52  typedef struct {
53        byte pwm_pin[2]; //Pulse Width Modulation pins
54        byte io_pin[2]; //Input/Output pins
55      }OUTPUT_PORT;
56
57  typedef struct {
58        byte fn;   //function
59        byte op;   //operand
60        byte port; //port #
61        byte port_type;
62        byte db; //daughterboard address
63        byte seq; //instruction sequence number
64        byte data; //databyte(s)
65        byte pin; //pin selector
66      }MicroCode;
67
68  //free pin analog 3
69
70  class X10ABOT_DB {
71
72    public:
73      X10ABOT_DB(byte logging);
74      X10ABOT_DB(byte db_address, byte logging);
75      ~X10ABOT_DB();
76
77      void receiveEvent(int numBytes);
78      void requestEvent();
79      void localRequest(byte * return_array);
80      void localReceive(byte * message, int numBytes);
81      static void receiveEvent_wrapper(int numBytes);
82      static void requestEvent_wrapper();
83      void execParse(MicroCode instr);
84      /**
85      * Logging Functions
86      **/
87      void i2cStatusLog(byte var);
88
89    private:
90      int _logging, _analog, _digital;
91      byte _rand_mid;
92      byte _lookup[6];
93
94
```

```
95
96  };
97
98    #endif
```

x10abot/X10ABOT_DB.cpp

```
1   /*
2     X10ABOT_DB.h - Library for  X10ABOT_DB code.
3     Created by Rohan A. Smith, January 31, 2012.
4     Released into the public domain.
5   */
6   #include <Wire.h>
7   #include "X10ABOT_DB.h" //include the declaration for this class
8
9   void* pt2Object;
10
11  //Used as the most fundamental level of coding
12  MicroCode instr;
13
14  //PWM pins are 3,5, 6,9, 10 & 11
15  //IO pin are: 2,4, 7,8, 12 & 13 (0,1 =>[rx,tx])
16  OUTPUT_PORT output[] = {
17    //{{PWM a, PWM b}, {IO a, IO b}}
18    {{10, 11}, {12, 13}},  //port 1
19    {{6, 9}, {7, 8}}, //port 2
20    {{3,5},{0,1}} //{0,1} =>[rx,tx])
21  };
22
23  //Analog pins are 0,1,2,3,4 or as Digital {14,15,16,17}
24  INPUT_PORT input[] = {
25    //{{IO a, IO b}, analog}
26    {{2, 4}, 0}, //port 1
27    {{16, 17}, 1} //port 2
28  };
29
30
31  //<<constructor>>
32  X10ABOT_DB::X10ABOT_DB(byte db_address, byte logging){
33    _logging = logging;
34    Wire.begin(db_address);                 // join i2c bus with address #4
35   //Wire.onRequest(requestEvent); // register event
36    Wire.onReceive(receiveEvent_wrapper); // register event
37    Wire.onRequest(requestEvent_wrapper);
38    _rand_mid = random(0, 20);
39  }
40
41  X10ABOT_DB::X10ABOT_DB(byte logging){
42    _logging = logging;
43  }
44
45
46  //<<destructor>>
```

```
47  X10ABOT_DB::~X10ABOT_DB(){/*nothing to destruct*/}
48
49  /*Byte 1:1111XXXX FUNCTION BYTE
50  Byte 1:XXXX1111 OPERAND BYTE
51  Byte 2:11111111 D.B. SELECTION
52  Byte 3:1111111X PORT SELECTION
53  Byte 3:XXXXXXX1 PIN SELECTION
54  Byte 4:11111111 INSTRUCTION SEQUENCE NUMBER
55  Byte 4+n 11111111 DATA BYTES; n> 0*/
56
57
58  /**
59  * Static wrapper method for the recieve event;
60  **/
61  void X10ABOT_DB::receiveEvent_wrapper (int numBytes){
62         // explicitly cast to a pointer to TClassA
63         X10ABOT_DB* mySelf = (X10ABOT_DB*) pt2Object;
64
65         // call member
66         mySelf->receiveEvent(numBytes);
67     }
68
69
70  /**
71  * Static wrapper method for the recieve event;
72  **/
73  void X10ABOT_DB::requestEvent_wrapper(){
74         // explicitly cast to a pointer to TClassA
75         X10ABOT_DB* mySelf = (X10ABOT_DB*) pt2Object;
76
77         // call member
78         mySelf->requestEvent();
79     }
80
81
82  /**
83  * Implementation of the Wire/I2C recieve event
84  **/
85
86  void X10ABOT_DB::requestEvent(){
87          Wire.write(_lookup, 7);
88  }
89
90  void X10ABOT_DB::localRequest(byte * return_array){
91    //Serial.print("_lookup0: ");Serial.println(_lookup[0]);
92    //Serial.print("_lookup1: ");Serial.println(_lookup[1]);
93    for (int j = 0; j <= 5; j++){
94      //Serial.print("_lookupj: ");Serial.println(_lookup[j]);
95      return_array[j] = _lookup[j];
96    }
97  }
98  /**
99  * Implementation of the Wire/I2C recieve event
```

```
100 **/
101
102 void X10ABOT_DB::receiveEvent(int numBytes)
103 {
104   byte fn_op, db, port;
105   while(0 < Wire.available()) // loop through all but the last
106   {
107
108     fn_op = Wire.read(); // receive FUNCTION & OPERATOR byte
109     instr.db = Wire.read(); // receive Daughter Board's # byte
110     port = Wire.read(); // receive Daughter Board's Port # byte
111     instr.seq = Wire.read();//receive instruction sequence number
112
113     byte op_mask = 0b00001111; // operator bitmask
114     byte pin_mask =0b00000001;  //pin bitmask
115
116     instr.fn   = fn_op  >> 4;
117     instr.port = (port&127) >> 1;
118     instr.port_type = port >> 7;
119     instr.op   = (fn_op & op_mask);
120     instr.pin  = (port & pin_mask);
121
122     //Serial.print("instr.fn: ");Serial.println(instr.fn, BIN);
123     //Serial.print("instr.op: ");Serial.println(instr.op, BIN);
124     //Serial.print("instr.db: ");Serial.println(instr.db, BIN);
125     //Serial.print("instr.port: ");Serial.println(port, BIN);
126     //Serial.print("instr.pt: ");Serial.println(instr.port, BIN);
127     //Serial.print("instr.pt_type: ");Serial.println(instr.port_type,
        ↪  BIN);
128     //Serial.print("instr.pn: ");Serial.println(instr.pin, BIN);
129
130     if (0<Wire.available())
131     {
132       while(0 < Wire.available()){ // loop through all but the last
133         //Serial.println("DATA");
134         instr.data = Wire.read(); // receive byte as a character
135         //Serial.print("instr.data: ");Serial.println(instr.data, BIN);
136         execParse(instr);      //Calls the function delegator
137       }
138     }
139     else{
140       execParse(instr);
141     }
142   }
143 }
144
145 void X10ABOT_DB::localReceive(byte * microcode, int numBytes)
146 {
147   byte fn_op, db, port;
148   if (numBytes>=DB_INSTR_HEADER_SIZE) // loop through all but the last
149   {
150
```

```
151       fn_op = microcode[DB_FUNCTION_OPERAND]; // receive FUNCTION & OPERAND
           ↪  byte
152       instr.db = microcode[DB_D_B_SELECTION]; // receive Daughter Board's #
           ↪  byte
153       port = microcode[DB_PORT_PIN]; // receive Daughter Board's Port # byte
154       instr.seq = microcode[DB_SEQ_NUM];//receive instruction sequence
           ↪  number
155
156       byte op_mask = 0b00001111; // operator bitmask
157       byte pin_mask =0b00000001;  //pin bitmask
158
159       instr.fn   = fn_op  >> 4;
160       instr.port_type = port >> 7;
161       instr.port = (port&255) >> 1;
162       instr.op   = (fn_op & op_mask);
163       instr.pin  = (port & pin_mask);
164       numBytes = numBytes - DB_INSTR_HEADER_SIZE;
165       //Serial.print("instr.fn: ");Serial.println(instr.fn, BIN);
166       //Serial.print("instr.op: ");Serial.println(instr.op, BIN);
167       //Serial.print("instr.db: ");Serial.println(instr.db, BIN);
168       //Serial.print("instr.pt: ");Serial.println(instr.port, BIN);
169       //Serial.print("instr.pn: ");Serial.println(instr.pin, BIN);
170       //Serial.print("instr.seq: ");Serial.println(instr.seq);
171
172       if (numBytes>DB_INSTR_HEADER_SIZE)
173       {
174         byte x = DB_INSTR_HEADER_SIZE;
175         while(0 < numBytes){ // loop through all but the last
176           //Serial.println("DATA");
177           instr.data = microcode[x]; // receive byte as a character
178           //Serial.print("instr.data: ");Serial.println(instr.data, BIN);
179           execParse(instr);      //Calls the function delegator
180           x++;
181           numBytes--;
182         }
183       }
184       else{
185         execParse(instr);
186       }
187     }
188   }
189
190   void X10ABOT_DB::execParse(MicroCode instr){  //byte fn, byte op, byte
       ↪  db, byte port, byte data)
191     int val;
192     switch( instr.fn )
193     {
194       case DB_FN_IO:{
195       //Serial.println("IO");
196
197         switch( instr.op )
198         {
199         //Initialise output port
```

```
200           case DB_OP_IO_HI:{
201             if(instr.port_type==1){
202               pinMode(input[instr.port].io_pin[instr.pin], OUTPUT);
203               digitalWrite(input[instr.port].io_pin[instr.pin], HIGH);
204             }else{
205               pinMode(output[instr.port].io_pin[instr.pin], OUTPUT);
206               digitalWrite(output[instr.port].io_pin[instr.pin], HIGH);
207             }
208             //Serial.print("PORT ");Serial.print(instr.port,
                ↪  DEC);Serial.print(" LED ");Serial.print(instr.op,
                ↪  DEC);Serial.print(" ON ");Serial.println(instr.pin, DEC);
209             break;
210           }
211
212           case DB_OP_IO_LOW:{
213             if(instr.port_type==1){
214               pinMode(input[instr.port].io_pin[instr.pin], OUTPUT);
215               digitalWrite(input[instr.port].io_pin[instr.pin], LOW);
216             }else{
217               pinMode(output[instr.port].io_pin[instr.pin], OUTPUT);
218               digitalWrite(output[instr.port].io_pin[instr.pin], LOW);
219             }
220             //Serial.print("PORT ");Serial.print(instr.port,
                ↪  DEC);Serial.print(" LED ");Serial.print(instr.pin,
                ↪  DEC);Serial.println(" OFF");
221             break;
222           }
223
224         //Initialise input port
225           case DB_OP_IO_INP:{
226       //case 2:
227             //Serial.println(input[instr.port].io_pin[instr.pin]);
228             int SensorData;
229             if(instr.port_type==1){
230               pinMode(input[instr.port].io_pin[instr.pin], INPUT);
231              SensorData = digitalRead(input[instr.port].io_pin[instr.pin]);
232             }else{
233               pinMode(output[instr.port].io_pin[instr.pin], INPUT);
234               SensorData =
                   ↪  digitalRead(output[instr.port].io_pin[instr.pin]);
235             }
236
237             //Serial.print(F("PortType:
                ↪  "));Serial.println(instr.port_type);
238             //Serial.print(F("PIN:
                ↪  "));Serial.println(input[instr.port].io_pin[instr.pin]);
239             //Serial.print(F("PIN:
                ↪  "));Serial.println(input[instr.port].io_pin[instr.pin]);
240             //Serial.print(F("SensorData: "));Serial.println(SensorData);
241             itoa(SensorData,(char*)_lookup,10);
242
243             byte z = sizeof(_lookup);
244             for(int j = z - 2; j >= 0; j--) {
```

```
245               _lookup[j+1] =  _lookup[j];
246             }
247             _lookup[0] = instr.seq;
248
249             break;
250           }
251         }
252
253         break;
254       }
255
256       case DB_FN_PWM:{
257         analogWrite(output[instr.port].pwm_pin[instr.pin], instr.data);
258         //Serial.print("PORT ");Serial.print(instr.port, DEC);
259         //Serial.print(" PWM("); Serial.print(instr.pin); Serial.print(")
            ↪  DATA="); Serial.println(instr.data);
260         break;
261       }
262
263       case DB_FN_SERIAL:{
264         break;
265       }
266
267       case DB_FN_ANALOG:{
268         int SensorData = analogRead(input[instr.port].analog);
269
270         itoa(SensorData,(char*)_lookup,10);
271
272         byte z = sizeof(_lookup);
273         for(int j = z - 2; j >= 0; j--) {
274           _lookup[j+1] =  _lookup[j];
275         }
276         _lookup[0] = instr.seq;
277         //Serial.write(_lookup, z);
278         /*for(int x=0; x<=z; x++){
279           Serial.print("_lookup[]): ");  Serial.println(_lookup[x]);
280           Serial.println("");
281         }*/
282         break;
283       }
284       //default:
285         //Serial.print("NONE: ");
286     }
287   }
```

x10abot/Sensor.cpp

```
1  #include "X10ABOT_MB.h"
2  //#include <MemoryFree.h>
3  /**
4   * Sensor Constructor
5   *
6   * @param  byte db_address   Sets the  daughterboard's address
```

```
7    * @param  byte port_number  Sets the  port_number on the daughterboard
     ↪  specified
8    */
9   //Sensor::Sensor(byte db_address, byte port_number)
10
11  /**
12   * Sensor Constructor
13   *
14   * @param  byte db_address  Sets the  daughterboard's address
15   * @param  byte port_number  Sets the  port_number on the daughterboard
     ↪  specified
16   * @param  byte pin_select  Selects either pin A(0) or B(1) on the
     ↪  selected port
17   */
18  /*Sensor::Sensor(byte db_address, byte port_number, byte pin_select){
19      _db = db_address;
20      _port = port_number;
21      _pin = pin_select;
22  }
23  */
24  /**
25   * Sensor Destructor
26   * no params
27   */
28  Sensor::~Sensor(){
29    /*nothing to destruct*/
30  }
31
32  //X10ABOT_MB sensor(LOGGING);
33
34  /**
35   * Sensor readDigital
36   *
37   * @param  byte power  Drives the Sensor in the negative or positive
     ↪  direction based on the power level. It operates btween -100 and +100.
38   */
39
40  byte Sensor::readDigital(){
41     byte x =  digitalIn(_db, _port, _pin);
42     return x;
43   }
44
45  void Sensor::writeDigitalHi(){
46   digitalOut(HI, _db, _port, _pin);
47  }
48
49  void Sensor::writeDigitalLo(){
50   digitalOut(LO, _db, _port, _pin);
51  }
52
53  int Sensor::readAnalog(){
54    return analog(_db,_port);
55   }
```

```
56
57 void Sensor::off(){}
58 void Sensor::readDigitalA(){}
59 void Sensor::readDigitalB(){}
```

x10abot/Microcode.cpp

```
1  #include "X10ABOT_MB.h"
2  /*
3  Byte 1:1111XXXX FUNCTION BYTE
4  Byte 1:XXXX1111 OPERAND BYTE
5  Byte 2:11111111 D.B. SELECTION
6  Byte 3:1111111X PORT SELECTION
7  Byte 3:XXXXXXX1 PIN SELECTION
8  Byte 4:11111111 INSTRUCTION SEQUENCE NUMBER
9  Byte 4+n 11111111 DATA BYTES; n> 0
10 */
11
12 /**
13  * Asserts a state to a Digital IO pin
14  *
15  * @param  byte  state  The state of the digital pin, HIGH(2), LOW(1),
    ↪  INPUT(0)
16  * @param  byte  db_address The daughterboard address between 7 and 120,
    ↪  0 for motherboard
17  * @param  byte  port_number the daughter board's port number connected
    ↪  to the device
18  * @param  byte  pin_select choose which of the I/O pins on the port to
    ↪  use A(0) or B(1)
19  * @return void
20  */
21 void X10ABOT_MB::digitalOut(byte state, byte db_address, byte
    ↪  port_number, byte pin_select){
22   byte microcode[] =
      ↪  {FN_IO+state,db_address,((port_number-1)<<1)+pin_select,incr_instr_seq()};
23   dispatch(microcode, sizeof(microcode));
24
25 }
26 byte X10ABOT_MB::digitalIn(byte db_address, byte port_number, byte
    ↪  pin_select){
27   byte seq_num = incr_instr_seq();
28   byte microcode[] =
      ↪  {FN_IO+IN,db_address,((port_number-1)<<1)+pin_select,seq_num};
29   dispatch(microcode, sizeof(microcode));
30   //delay(50);
31    return requestHandler(microcode, 2,seq_num);
32 }
33 /**
34  * Asserts a PWM signal to a pin
35  *
36  * @param  byte  pwm_select choose which of the PWM pins on the port to
    ↪  use A(0) or B(1)
```

```
37   * @param  byte  db_address The daughterboard address between 7 and 120,
     ↪  0 for motherboard
38   * @param  byte  port_number the daughter board's port number connected
     ↪  to the device
39   * @param  byte  duty_cycle specify the duty cycle as a number between 0
     ↪  and 255
40   * @return void
41   */
42  void X10ABOT_MB::pwm(byte pwm_select, byte db_address, byte port_number,
     ↪  byte duty_cycle){
43    byte microcode[] =
       ↪  {FN_PWM,db_address,((port_number-1)<<1)+pwm_select,incr_instr_seq(),duty_cycle};
44    dispatch(microcode, sizeof(microcode));
45  }
46
47  /**
48   * Reads the state from an Analogue pin
49   *
50   * @param  byte  db_address The daughterboard address between 7 and 120,
     ↪  0 for motherboard
51   * @param  byte  port_number the daughter board's port number connected
     ↪  to the device
52   * @return int
53   */
54  int X10ABOT_MB::analog(byte db_address, byte port_number){
55    byte seq_num = incr_instr_seq();
56    byte microcode[] =   {FN_ANALOG,db_address,((port_number-1)<<1),seq_num
       ↪  };
57    dispatch(microcode, sizeof(microcode));
58    //delay(50);
59    return requestHandler(microcode, 6,seq_num);
60  }
```

x10abot/Actuator.cpp

```
1   //#include "Actuator.h"
2   #include "X10ABOT_MB.h"
3   /**
4    * Actuator Constructor
5    *
6    * @param  byte db_address   Sets the  daughterboard's address
7    * @param  byte port_number  Sets the  port_number on the daughterboard
     ↪  specified
8    */
9   //Actuator::Actuator(byte db_address, byte port_number)
10
11  /**
12   * Actuator Constructor
13   *
14   * @param  byte db_address  Sets the  daughterboard's address
15   * @param  byte port_number  Sets the  port_number on the daughterboard
     ↪  specified
```

```
16   * @param  byte port_number  Selects either pin A(0) or B(1) on the
     ↪  selected port
17   */
18  /*Actuator::Actuator(byte db_address, byte port_number, byte pin_select){
19      _db = db_address;
20      _port = port_number;
21      _pin = pin_select;
22  }
23  */
24  /**
25   * Actuator Destructor
26   * no params
27   */
28  Actuator::~Actuator(){
29    /*nothing to destruct*/
30  }
31
32  X10ABOT_MB actuator(LOGGING);
33
34  /**
35   * Actuator run
36   *
37   * @param  byte power  Drives the actuator in the negative or positive
     ↪  direction based on the power level. It operates btween -100 and +100.
38   */
39
40  void Actuator::run(byte power){
41    if (power>100)
42    {
43      power=100;
44    }else if (power<(-100))
45    {
46      power = -100;
47    }
48
49    if (power>0)
50    {
51      actuator.pwm(A, _db, _port, 255*power/100);
52      actuator.digitalOut(HI,_db,_port,A);
53      actuator.digitalOut(LO,_db,_port,B);
54    }
55    else{
56      actuator.pwm(A, _db, _port, 255*power/100);
57      actuator.digitalOut(LO,_db,_port,A);
58      actuator.digitalOut(HI,_db,_port,B);
59    }
60  }
61
62  void Actuator::on(byte power){
63    actuator.digitalOut(HI,_db,_port,A);
64    actuator.digitalOut(HI,_db,_port,B);
65  }
66
```

```
67  void Actuator::on(){
68    //Serial.println("Motor ON!");
69    actuator.digitalOut(HI,_db,_port,A);
70    actuator.digitalOut(HI,_db,_port,B);
71  }
72
73  void Actuator::off(){
74    actuator.digitalOut(LO,_db,_port,A);
75    actuator.digitalOut(LO,_db,_port,B);
76  }
77
78  void Actuator::on_a(){
79    actuator.digitalOut(HI,_db,_port,A);
80  }
81
82  void Actuator::off_a(){
83    actuator.digitalOut(LO,_db,_port,A);
84  }
85
86  void Actuator::on_b(){
87    actuator.digitalOut(HI,_db,_port,B);
88  }
89
90  void Actuator::off_b(){
91    actuator.digitalOut(LO,_db,_port,B);
92  }
93
94  /*
95  * Sets digital pin values to High or Low
96  */
97  void Actuator::setDigital(byte pin_a, byte pin_b){
98    actuator.digitalOut(pin_a,_db,_port,A);
99    actuator.digitalOut(pin_b,_db,_port,B);
100 }
101
102 void Actuator::pwm_a(byte power){
103   actuator.pwm(A, _db, _port, 255*power/100);
104 }
105
106 void Actuator::pwm_b(byte power){
107   actuator.pwm(B, _db, _port, 255*power/100);
108 }
109
110 void Actuator::stop(){
111   actuator.digitalOut(LO,_db,_port,B);
112   delay(100);
113   actuator.digitalOut(HI,_db,_port,B);
114   delay(100);
115 }
```

```
#include "../X10ABOT_MB.h"
#include <math.h>
class Thermistor: public Sensor
{
  private:
    byte _db, _port;
  public:
    Thermistor(byte db_address, byte port_number):Sensor(_db, _port){
      _db = db_address;
      _port = port_number;
    }
    ~Thermistor(){};
    double readThermistorCelcius() {
        //The analog micrcode requests the raw sensor value
        int RawADC = analog(_db,_port);
        double Temp;
        Temp = log(((10240000/RawADC) - 10000));
        Temp =
         ↪  1/(0.001129148+(0.000234125+(0.0000000876741*Temp*Temp))*Temp);
        Temp = Temp - 273.15;            // Convert Kelvin to Celcius
        return Temp;
    }
    double readThermistorFarenheit() {
    // Convert Celcius to Fahrenheit
        return (readThermistorCelcius()* 9.0)/ 5.0 + 32.0;
    }
};
```

x10abot/Tests_and_Logs_MB.cpp

```
#include "X10ABOT_MB.h"

/**
 * Monitors the stats of I2C transmissions
 *
 * @param  byte  var  Return status of Transmission
 * @return void
 */
void X10ABOT_MB::i2cStatusLog(byte var){
  if(_logging){
    switch (var) {
      case 0:
      Serial.println("I2C 0:success");
      break;
      case 1:
      Serial.println("I2C 1:data too long to fit in transmit buffer");
      break;
      case 2:
      Serial.println("I2C 2:received NACK on transmit of address");
      break;
      case 3:
      Serial.println("I2C 3:received NACK on transmit of data");
      break;
```

```
24        case 4:
25        Serial.println("I2C 4:other error");
26        break;
27        default:
28        Serial.println("I2C 4:other error");
29      }
30    }
31  }
32
33  void X10ABOT_MB::dispatchDataLog(byte * microcode, int size){
34    if(_logging){
35      Serial.println(microcode[0], BIN);
36      Serial.println(microcode[1], BIN);
37      Serial.println(microcode[2], BIN);
38      //Serial.println(microcode[3]);
39      Serial.println("---------------");
40
41    }
42  }
43
44
45  void X10ABOT_MB::test_function(){
46    //Serial.println("TEST FUNCTION");
47    //Serial.println(OP_IO_HI, BIN);
48    //Serial.print("Added: ");Serial.println((byte)(FN_IO+OP_IO_HI), BIN);
49    byte test_pattern[] =   {FN_IO+OP_IO_HI,0,PORT_1+PIN_B,11};
      ↪  //1-1(17)-9-1
50    byte test_pattern2[] =  {FN_IO+OP_IO_LOW,0,PORT_1+PIN_B,8}; //1-2-9-2
51    byte test_pattern3[] =  {FN_PWM+OP_NOP,0,PORT_1+PIN_B, 255*(20.0/100)};
      ↪  //1-2-9-2
52    byte test_pattern4[] =  {FN_PWM+OP_NOP,0,PORT_1+PIN_B, 255*(80.0/100)};
      ↪  //1-2-9-2
53    byte test_pattern6[] =  {FN_IO+OP_IO_LOW,9,PORT_1+PIN_B,8}; //1-2-9-2
54    byte test_pattern7[] =   {FN_IO+OP_IO_HI,9,PORT_1+PIN_B,11};
      ↪  //1-1(17)-9-1
55  //Serial.write("size:"); Serial.println(sizeof(test_pattern));
56    dispatch(test_pattern, 3);
57    dispatch(test_pattern7, 3);
58
59    // fade out from max to min in increments of 5 points:
60    for(int fadeValue = 0 ; fadeValue <= 255; fadeValue +=5) {
61      // sets the value (range from 0 to 255):
62      //Serial.write("fadeValue: "); Serial.println(sizeof(fadeValue));
63      byte test_pattern5[] =  {FN_PWM+OP_NOP,0,PORT_1+PIN_B, fadeValue};
64      byte test_pattern5a[] =  {FN_PWM+OP_NOP,9,PORT_1+PIN_B, fadeValue};
65      //Serial.write("pre dispatch: ");
66    dispatch(test_pattern5, 4);
67    dispatch(test_pattern5a, 4);
68    //Serial.write("post dispatch: ");
69      // wait for 30 milliseconds to see the dimming effect
70      delay(30);
71    }
72    //Serial.println("WAXOFF!");
```

```
  //delay(1000);
  dispatch(test_pattern2, 3);
  dispatch(test_pattern6, 3);
  for(int fadeValue = 255 ; fadeValue >= 0; fadeValue -=5) {
    // sets the value (range from 0 to 255):
    byte test_pattern5[] =  {FN_PWM+OP_NOP,0,PORT_1+PIN_B, fadeValue};
    byte test_pattern5a[] =  {FN_PWM+OP_NOP,9,PORT_1+PIN_B, fadeValue};
    dispatch(test_pattern5, 4);
    dispatch(test_pattern5a, 4);
    // wait for 30 milliseconds to see the dimming effect
    delay(30);
  }
  //delay(1000);
  //Serial.println("WAXON!");
}
```

x10abot/README.md

````
X10ABOT_DB
==========

Daughterboard Library for the X10ABOT Robotics Platform

Example Code
------------
```C++
    #include <X10ABOT_DB.h>

    #define LOGGING 1
    #define DB_ADDRESS 9
    #include <Wire.h>

    X10ABOT_DB bot(DB_ADDRESS, LOGGING);
    void setup(){
      Serial.begin(9600);
    }
    void loop(){
    }
```
X10ABOT_MB
==========

Motherboard Library for the X10ABOT Robotic Platform
To test this library, run the example code below
Example code:
-------------
    #include <X10ABOT_MB.h>
    #include <X10ABOT_DB.h>
    #include <Wire.h>

    Actuator motor1(0,1);  //DC Motor, daughterboard 0, output port 1
    Sensor force(0,1);  //Analog Force Sensor, daughterboard 0, input
     ↪  port 1
````

```
35      void setup(){
36        Serial.begin(9600);
37      }
38
39      void loop(){
40        motor1.on(100);
41        Serial.println(force.getAnalog());
42    }
```

x10abot/keywords.txt

```
1  ########################################
2  # Syntax Coloring Map For X10ABOT
3  ########################################
4
5  ########################################
6  # Datatypes (KEYWORD1)
7  ########################################
8
9  X10ABOT_MB KEYWORD1
10
11 ########################################
12 # Methods and Functions (KEYWORD2)
13 ########################################
14
15 on  KEYWORD2
16 off KEYWORD2
17 blink KEYWORD2
18 dash  KEYWORD2
19 dot KEYWORD2
20
21 ########################################
22 # Constants (LITERAL1)
23 ########################################
24 HI LITERAL1
25 LO LITERAL1
26 IN LITERAL1
27
28
29 ########################################
30 # COLOR CODE
31 ########################################
32 # KEYWORD1 Classes, datatypes, and C++ keywords
33 # KEYWORD2 Methods and functions
34 # KEYWORD3 setup and loop functions, as well as the Serial keywords
35 # LITERAL1 Constants
36 # LITERAL2 Built-in variables (unused by default)
37 ########################################
```

All source code listed here as well as the complete `X10ABOT` Library can be found at https://github.com/frazras/x10abot at commit 4a90414b6ea44620e459ea11ba336f0dbdc099f1.